From Modelling to Systematic Deployment of Distributed Active Objects – Extended Version

Ludovic Henrio, Justine Rochas

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES DE SOPHIA ANTIPOLIS
UMR 7271

From Modelling to Systematic Deployment of Distributed Active Objects – Extended Version

Ludovic Henrio, Justine Rochas
EQUIPE Scale

Rapport de Recherche

04-2016

Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis (I3S) - UMR7271 - UNS CNRS
2000, route des Lucioles — Les Algorithmes - bât. Euclide B — 06900 Sophia Antipolis — France
http ://www.i3s.unice.fr

Membre de UNIVERSITÉ CÔTE D'AZUR

From Modelling to Systematic Deployment of Distributed Active Objects – Extended Version

Ludovic Henrio [1], Justine Rochas [2]

EQUIPE Scale
04-2016 - 51 pages

Abstract : In the context of the expansion of actors and active objects, we are still facing a gap between the safety guaranteed by modelling and verification languages and the efficiency of distributed middlewares. In this paper, we reconcile two active object-based languages, ABS and ProActive, that respectively target the aforementioned goals. We compile ABS programs into ProActive, making possible to benefit from the strengths of both languages, while requiring no modification on the source code. After introducing the translational semantics, we establish the properties and the correctness of the translation. Overall, this paper presents an approach to running different active object models in distributed environments, and more generally studies the implementation of programming languages based on active objects.

Key-words : active object, distribution, programming model, programming language, semantics, parallelism, concurrency

1. Laboratoire I3S – CNRS – ludovic.henrio@cnrs.fr
2. Laboratoire I3S – Université Nice Sophia Antipolis – justine.rochas@unice.fr

From Modelling to Systematic Deployment of Distributed Active Objects – Extended Version

Résumé : Dans le contexte de l'expansion des acteurs et des objets actifs, il existe toujours un écart visible entre la sûreté garantie par les langages de modélisation et de vérification, et l'efficacité des intergiciels distribués. Dans cet article, nous proposons de réconciler deux langages de programmation à objets actifs, ABS et ProActive, qui ciblent respectivement les deux objectifs précédents. En compilant des programmes ABS vers des programmes ProActive, il est possible de bénéficier des avantages des deux langages, sans requérir de modifications dans le code source ABS. Après avoir introduit la sémantique translationnelle, nous établissons la correction de la traduction. Plus généralement, cet article étudie l'implantation efficace des langages de programmation à objets actifs.

Mots-clefs : objet actif, distribution, modèle de programmation, langage de programmation, sémantique, parallélisme, concurrence

From Modelling to Systematic Deployment of Distributed Active Objects – Extended Version

Ludovic Henrio and Justine Rochas

Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271 06900 Sophia Antipolis, France
ludovic.henrio@cnrs.fr, justine.rochas@unice.fr

Abstract. In the context of the expansion of actors and active objects, we are still facing a gap between the safety guaranteed by modelling and verification languages and the efficiency of distributed middlewares. In this paper, we reconcile two active object-based languages, ABS and ProActive, that respectively target the aforementioned goals. We compile ABS programs into ProActive, making possible to benefit from the strengths of both languages, while requiring no modification on the source code. After introducing the translational semantics, we establish the properties and the correctness of the translation. Overall, this paper presents an approach to running different active object models in distributed environments, and more generally studies the implementation of programming languages based on active objects.

1 Introduction

Writing distributed and concurrent applications is a challenging task. In distributed environments, the absence of shared memory makes information sharing more difficult. In concurrent environments, data sharing is easy but shared data must be manipulated with caution. Several languages and tools have been developed to handle those two programming challenges and make distributed and concurrent systems safe by construction. Among them, the active object programming model [16] helps building safe multi-core applications in object-oriented programming languages. The active object model derives from the actor model [1] that is particularly regaining popularity with Scala [11] and Akka[1]. Such models are natively adapted to distribution because entities do not share memory and behave independently from each other.

There exist now several programming languages implementing and enhancing in various ways the active object and actor models. In particular, emerging active object languages, like the Abstract Behavioral Specification language [14] (hereafter ABS), provide various programming abstractions or static guarantees that help the developer designing and implementing robust distributed systems. Among existing implementations of active objects, ProActive[2] is a Java middleware implementing multi-threaded active objects that provides a holistic support

[1] http://akka.io
[2] http://proactive.inria.fr/

for deployment and execution of active objects on distributed infrastructures. This paper reconciles cooperative active object languages by translating their main concurrent paradigms into ProActive, thus benefiting from its support for deployment. We illustrate our approach on ABS, which has a wide support for modelling and verification. We translate all the concurrent object layer of ABS into ProActive. We also introduce in this paper MultiASP, a formal language that models ProActive, in order to verify the translation.

Beyond the generic high-level approach to cross-translating active object languages, the practical contribution of this paper is a ProActive backend for ABS, that automatically translates an ABS application into a distributed ProActive application. As a result, the programmer can design and verify his program using the powerful toolset of ABS, and then generate efficient distributed Java code that runs with ProActive. The proof of correctness of the translation ensures the equivalence of execution in terms of the operational semantics. Consequently, it guarantees that the verified properties dealing with the program behaviour (e.g. absence of deadlocks, typing properties) will still be valid. Our approach requires no change in the ABS code except the minimal (required) deployment information. Overall, our contribution can be summarised in four points:

- We analyse existing active object programming paradigms in Section 2.
- We provide MultiASP, a class-based semantics of the multi-threaded active objects featured in ProActive in Section 3.
- We present a systematic strategy to translate active objects with cooperative scheduling into ProActive, and present more specifically the ProActive backend for ABS in Section 4. The translation is formalised in Section 5.
- We prove translation equivalence in Section 6 and highlight similarities and differences between active object models. In particular the proof of equivalence reveals intrinsic differences between explicitly typed futures and transparent first-class futures.

2 Background and Related Works

The actor model was one of the first to schematically consider concurrent entities evolving independently and communicating via asynchronous messages. Later on, active objects have been designed as the object-oriented counterpart of the actor model. The principle of active objects is to have a thread associated to them. We call this notion *activity*: a thread together with the objects managed by this thread. Objects from different activities communicate with remote method invocations: when a method is invoked on a remote active object, this creates a *request* in the remote activity; the invoker continues its execution while the invoked active object serves the request asynchronously. Requests wait in a *request queue* until they are executed. In order to allow the invoker to continue execution, a placeholder for the expected result is created, known as *future* [9]: an empty object that will later be filled by the result of the request. When the value of a future is known, we say that it is *resolved*.

2.1 Design choices for active object-based languages

Implementing active objects raises the three following questions:

How are objects associated to activities? In uniform active object models, all objects are active and have their own execution thread (e.g. Creol [15]). This model is distinguished from non uniform active object models which feature active and passive objects (e.g. ASP [6]). Each passive object is a normal object not equipped with any thread nor request queue; there is no race condition on the access to passive object because each of them is accessible by a single active object. In practice, non uniform active object models are more scalable, but they are trickier to formalise than uniform active object models. A trade-off between those two models appeared with JCoBox [18] that introduced the active object group model, where all objects are accessible from any object, but where objects of the same group share the same execution thread.

How are requests scheduled? The way requests are executed in active objects depends on the threading model used. In the original programming model, active objects are mono-threaded. With cooperative scheduling like in Creol, requests in execution can be paused on some condition (e.g. awaiting on the resolution of a future), letting another request progress in the meantime. In all cooperative active object languages, while no data race is possible, interleaving of the different request services (triggered by the different release points) makes the behaviour more difficult to predict than for the mono-threaded model. Still, the previous models are inefficient on multi-cores and can lead to deadlocks due to reentrant calls and/or inadequately placed release points. Newest active object models like multiactive objects [12] and Encore [5] feature controlled multi-threading. Such active object models succeed in maximising local parallelism while avoiding communication overhead, thanks to shared memory between the different threads [12]. Also, controlled multi-threading prevents many deadlocks in active object executions.

Is the programmer aware of distributed aspects? Existing implementations of active objects either choose to hide asynchrony and distribution or, on the contrary to use an explicit syntax for handling asynchronous method calls and to use an explicit type for handling futures. This makes the programmer aware of where synchronisation occurs, but consequently requires more expertise. The choice of transparency also impacts the language possibilities, like future reference transmission: it is easier to transmit futures between active objects when no specific future type is used, and the programmer does not have to know how many future indirections have to be unfolded to get the final value.

2.2 Overview of active object-based languages

Creol [15] is a uniform active object language that features cooperative scheduling based on `await` operations that can release the execution thread. In this language, asynchronous invocations and futures are explicit, and futures are not transmitted between activities. De Boer et al. formalised such futures based on

g::=b	$\mid\ x?\ \mid\ g \wedge g'$	guard
s::=`skip`	$\mid\ x = z\ \mid$ suspend $\mid$ await g	statement
	$\mid$ `return` $e\ \mid$ `if` $e\ \{s\}$ `else` $\{s\}\ \mid\ s$ `;` s	
z::=e	$\mid\ e.\mathtt{m}(\overline{e})\ \mid\ e!\mathtt{m}(\overline{e})\ \mid$ `new` $[cog]C(\overline{e})\ \mid\ x$.get	expression with side effect
e::=v	$\mid\ x\ \mid$ `this` $\mid$ *arithmetic-bool-exp*	expression
v::=`null`	$\mid$ *primitive-val*	value

Fig. 1: Class-based syntax of the concurrent object layer of ABS. Field access is restricted to current object (`this`).

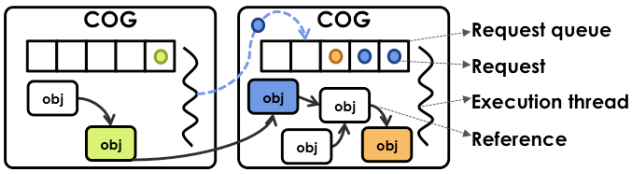


Fig. 2: An example of ABS program execution

Creol in [4]. Overall, explicit future access, explicit release points, and explicit asynchronous calls make Creol rich and precise but also more difficult to program than the languages featuring more transparency.

JCoBox [18] is an active object programming model implemented in a language based on Java. It has an object group model, called CoBox, and also features cooperative scheduling. In each CoBox, a single thread is active at a time; it can be released using `await()`. JCoBox better addresses practical aspects than Creol: it is integrated with Java and the object group model improves thread scalability, however JCoBox does not support distributed execution. Thread interleaving is similar and has the same advantages and drawbacks as in Creol.

AmbientTalk [7] is an object-oriented distributed programming language that can execute on the JVM. One original aspect of AmbientTalk is that a future access is a non-blocking operation: it is an asynchronous call that returns another future; the call will be performed when the invoked future is resolved. The AmbientTalk future model forces two activities to coordinate only through callbacks. This inversion of control has the advantage to avoid deadlocks but also breaks the program into independent procedures where sequences of instructions are difficult to enforce.

ABS [14] is an active object-based language that targets modelling of distributed applications. The fragment of the ABS syntax regarding the concurrent object layer is shown on Figure 1. ABS has an object group model, like JCoBox, based on the notion of concurrent object group (hereafter COG). Asynchronous method calls and futures are explicit:

```
1 Fut<V> future = object!method();
```

Figure 2 pictures an ABS configuration with a request sending between COGs. Requests are scheduled in a cooperative manner thanks to the `await` keyword,

inspired from Creol and JCoBox and used as follows:

```
await future?;  await a > 2 && b < 3;
```

In those examples, the execution thread is released if the future is not resolved or if the condition is not fulfilled. ABS also features a `get` accessor to retrieve a future's value; it blocks the execution thread until the future is resolved:

```
V v = future.get;
```

The ABS tool suite[3] provides a wide variety of static verification engines that help designing safe distributed and concurrent applications. Those engines include a deadlock analyser [10], a resource and cost analyser for cloud environments [2], and general program properties verification with the ABS-Key tool [8]. The ABS tool suite also includes a frontend compiler and several backend translators into various programming languages. The Java backend for ABS translates ABS programs into concurrent Java code that runs on a single machine. The Haskell backend for ABS [3] performs the translation into distributed Haskell code. The ABS semantics is preserved thanks to the thread continuation support of Haskell, which is not supported on the JVM.

ASP and ProActive. Asynchronous Sequential Processes (ASP) [6] is a mono-threaded active object programming language that has a non-uniform object model. In ASP, active objects are transparent to the programmer and futures are created and manipulated implicitly. A wait-by-necessity is triggered upon access to an unresolved future. Futures are first class: they are transparently passed and updated across activities. ProActive is the Java library that implements ASP. ProActive is a middleware that supports application deployment on distributed infrastructures such as clusters, grids and clouds. The program below creates explicitly an active object using `newActive` instead of `new`. The variable `v` stores an implicit future that is the result of a (transparent) asynchronous call.

```
T t = PAActiveObject.newActive(T.class, parameters, node);
V v = t.bar();
o.foo(v);   // does not block even if v is unresolved (o is any active or passive object)
v.foobar(); // blocks if v is unresolved
```

Recently, ProActive integrated multiactive objects [12] to enable multi-threaded request processing. MultiASP, presented in the next section, is an update of ASP and thus formalises the new version of ProActive. In practice, a programmer declares which requests of an active object can safely be executed in parallel, namely which requests are *compatible*, as shown in the following example:

```
@Group(name="group1", selfCompatible=true)
@Group(name="group2", selfCompatible=false)
@Compatible({"group1", "group2"})
public class MyClass {
  @MemberOf("group1")   public ... method1(...) { ... }
  @MemberOf("group2")   public ... method2(...) { ... }
}
```

In this example, a request for `method1` can be executed at the same time as a request for `method2`, but two requests for `method2` cannot be executed at the

[3] http://abs-models.org/

same time. With similar annotations, it is also possible to set a limit on the number of threads running in parallel [13]. The limit can be applied in two ways: a hard limit restrains the overall number of threads whereas a soft limit only counts threads that are not in wait-by-necessity.

Encore. Encore [5] is an active object-based parallel language currently in development. Encore features active and passive objects but even if passive objects are private by default, they can be shared at different scales depending on qualifying keywords. Asynchronous calls are transparent for active objects (by default) but futures are explicit, using a dedicated type. Finally, an active object has a single thread of execution by default, but parallelism is automatically created by attaching callbacks to future updates and using parallel combinators.

2.3 Positioning of this work

The reason why there are many different implementations of the active object programming model is to better fit particular objectives, from reasoning about programs to optimised program execution. Implementations that focus on the deployment of real-world systems comply to constraints related to existing execution platforms and languages. They are mostly used by programmers interested in the performance of the application. ProActive and Encore typically fit in this category. On the other side, some active object languages target verification and proof of programs, but have not been originally designed for efficient execution, like typically ABS and Creol. They are massively used and developed by academics and less constrained by existing execution platforms.

We give a proven translation of ABS programs into ProActive code in order to reconcile both domains: verified applications also have the right to be run efficiently. We also study the generalisation of our approach to other active object languages. Overall, our objective is to show that generic active object abstractions can be correctly encoded with different active object implementations.

3 Class-based Semantics of MultiASP

We start by introducing the semantics of MultiASP[4], the calculus representing ProActive and multiactive objects. Unlike the preliminary formalisation of multiactive objects in [12], we present here a class-based formalisation and the formalisation of threading policies. MultiASP is an imperative programming language and its syntax is close to the one of ABS.

Syntax of MultiASP. Figure 3 shows the static syntax of MultiASP. A program is made of classes and a main method. $\overline{x}$ denotes local variables in method bodies and object fields in class declarations. There are two ways to create an object: *new* creates a new object in the current activity, and *newActive* creates a new active object. $e.\mathtt{m}(\overline{e})$ is the generic method invocation, there is no syntactic distinction between local and remote (asynchronous) invocations. Similarly, as

[4] Formalised in Isabelle/HOL: `www-sop.inria.fr/members/Ludovic.Henrio/misc.html`

$P ::= \overline{C}\ \{\, \overline{x}\ \texttt{;}\ \ s \,\}$		program
$S ::= \texttt{m}(\overline{x})$		method signature
$C ::= \texttt{class C}(\overline{x})\ \{\, \overline{x}\ \overline{M} \,\}$		class
$M ::= S\{\, \overline{x}\ s \,\}$		method definition
$s ::= \texttt{skip} \mid x = z \mid \texttt{return}\ e \mid s\ \texttt{;}\ s$		statement
$z ::= e \mid e.\texttt{m}(\overline{e}) \mid \texttt{new C}(\overline{e}) \mid \texttt{newActive C}(\overline{e})$		expression with side effects
$e ::= v \mid x \mid \texttt{this} \mid \textit{arithmetic-bool-exp}$		expression
$v ::= \texttt{null} \mid \textit{primitive-val}$		value

Fig. 3: Class-based static syntax of MultiASP

$$v ::= o \mid \alpha \mid \dots \qquad \textit{Storable} ::= [\overrightarrow{x \mapsto v}] \mid v \mid f$$
$$\textit{elem} ::= \textsc{fut}(f, v, \sigma) \mid \textsc{fut}(f, \bot) \mid \textsc{act}(\alpha, o, \sigma, p, Rq) \qquad \sigma ::= \overrightarrow{o \mapsto \textit{Storable}}$$
$$\textit{cn} ::= \overline{\textit{elem}} \qquad q ::= (f, m, \overline{v})$$
$$E ::= \{\ell \mid s\} \qquad Rq ::= \varnothing \mid q :: Rq$$
$$F ::= E \mid E :: F \qquad \ell ::= \texttt{this} \mapsto v, \overrightarrow{x \mapsto v}$$
$$p ::= \overrightarrow{q \mapsto F} \qquad s ::= x = \bullet \mid \dots$$

Fig. 4: Runtime syntax of MultiASP

synchronisation on futures is transparent and handled with wait-by-necessity, there is no particular syntax for interacting with a future. A special variable `this` exists for accessing the current object.

Semantics of MultiASP. MultiASP semantics is defined as a transition relation between configurations, noted *cn*, and for which the runtime syntax is displayed in Figure 4. At runtime, the dynamic configuration of a MultiASP program consists of a set of activities and a set of futures. The transition relation uses three infinite sets: *object locations* in the local store, ranged over by o, o', $\cdots$; *active objects names*, ranged over by α, β, $\cdots$; and *future names*, ranged over by f, f', $\cdots$. *Activities* are of the form $\textsc{act}(\alpha, o, \sigma, p, Rq)$ where α is an activity name; o is the location of the active object in σ; σ is a *local store* mapping object locations to storable values; p is a set of *requests currently served* (a mapping from requests to their thread F); and Rq is a FIFO *request queue* of requests awaiting to be served. A thread is a stack of methods being executed, and each *method execution* E consists of *local variables* ℓ and *statement* s to execute. The first method of the stack is the one that is executing, the others have been put in the stack due to local synchronous method calls. ℓ is a mapping from local variables (including `this`) to runtime values. A configuration also contains *future binders*. They are of two forms: $\textsc{fut}(f, \bot)$, meaning that the value for the future has not been computed yet, and $\textsc{fut}(f, v, \sigma)$, when the *reply value* is known; if it is an object (and not a static value), then v will be its location in the store σ.

An object o is fresh if it does not exist in the store in which it is added. Similarly, a future or an activity name is fresh if it does not exist in the current configuration. *Runtime values* ($v, \cdots$) can be either static values, object locations, or active object names. An object is a mapping from field names to their values, denoted $[\overrightarrow{x \mapsto v}]$. We denote mappings by $\overrightarrow{\_ \mapsto \_}$, and use union $\cup$ (resp. disjoint union $\uplus$) over mappings. Mapping updates are of the form $\sigma[x \mapsto v]$. *dom* returns the domain of a mapping. *Storable values* are objects, futures, or runtime values.

$$
\begin{array}{l}
\mathrm{serialise}(o,\sigma) = \\
\quad (o \mapsto \sigma(o)) \cup \mathrm{serialise}(\sigma(o),\sigma) \\
\mathrm{serialise}([\overrightarrow{x \mapsto v}],\sigma) = \\
\quad \bigcup_{v' \in \overline{v}} \mathrm{serialise}(v',\sigma) \\
\mathrm{serialise}(f,\sigma) = \\
\quad \mathrm{serialise}(\alpha,\sigma) = \\
\quad \mathrm{serialise}(\texttt{null},\sigma) = \varnothing \\
\mathrm{serialise}(\textit{primitive-val},\sigma) = \varnothing
\end{array}
$$

Fig. 5: Serialisation

$$
\begin{array}{rcll}
[\![\textit{primitive-val}]\!]_{(\sigma+\ell)} & \triangleq & \textit{primitive-val} & \\
[\![f]\!]_{(\sigma+\ell)} & \triangleq & \bot & \\
[\![\alpha]\!]_{(\sigma+\ell)} & \triangleq & \alpha & \\
[\![\texttt{null}]\!]_{(\sigma+\ell)} & \triangleq & \texttt{null} & \\
[\![x]\!]_{(\sigma+\ell)} & \triangleq & [\![\ell(x)]\!]_{(\sigma+\ell)} & \text{if } x \in \mathrm{dom}(\ell) \\
[\![x]\!]_{(\sigma+\ell)} & \triangleq & [\![\ell(\texttt{this})(x)]\!]_{(\sigma+\ell)} & \text{if } x \notin \mathrm{dom}(\ell) \\
[\![o]\!]_{(\sigma+\ell)} & \triangleq & o & \text{if } \sigma(o) = f \text{ or } \sigma(o) = [\overrightarrow{x \mapsto v}] \\
[\![o]\!]_{(\sigma+\ell)} & \triangleq & [\![\sigma(o)]\!]_{(\sigma+\ell)} & \text{else}
\end{array}
$$

Fig. 6: Evaluation function

The following auxiliary functions are used in the semantic rules: $[\![e]\!]_{(\sigma+\ell)}$ returns the value of e by computing the arithmetic and boolean expressions and by retrieving the values stored in σ or ℓ; the evaluation function is displayed in Figure 6. If the value of e is a reference to a location in the store, it follows references recursively; it only returns a location if the location points to an object or a future. $[\![\overline{e}]\!]_{(\sigma+\ell)}$ returns the tuple of values of $\overline{e}$. $\mathit{fields}(\texttt{C})$ returns fields as defined in the class declaration `C`. *bind* initialises method execution: $\mathrm{bind}(o,\texttt{m},\overline{v'}) = \{y \mapsto v', \overrightarrow{z \mapsto \texttt{null}}, \texttt{this} \mapsto o \mid s\}$, where the arguments of method `m`, typed in the class of o, are $\overline{y}$, and where the method body is $\{\overline{z}; s\}$. *ready* is a predicate deciding whether a request q in the queue Rq is ready to be served: $\mathit{ready}(q,p,Rq)$ is *true* if q is compatible with all requests in p (requests currently served by the activity) and with older requests in Rq. Serialisation reflects the communication style happening in Java RMI; it ensures that each activity has a single entry point: the active object. Consequently, all references to passive objects are serialised when communicated between activities, so that they are always handled locally. $\mathit{serialise}(o,\sigma)$ marks and copies the objects referenced from o to deeply serialise, recursively; it returns a new store made of all the objects that are referenced by o. *serialise* is defined as the mapping verifying the constraints of Figure 5. $\mathit{rename}_\sigma(\overline{v},\sigma')$ renames the object locations appearing in $\overline{v}$ and σ', making them disjoint from the object locations of σ; it returns a renamed set of values $\overline{v'}$ and a store σ''.

Figure 7 shows the part of MultiASP semantics that regards active object execution. Rules involving classical objects, namely object creation, field assignment, passive invocation, and local return of method call have been removed due to space limitation. The full MultiASP semantics can be found in Appendix B. In all cases, rules only show activities and futures involved in the current reduction. SERVE picks the first request that is ready in the queue (compatible with executing requests and with older requests in the queue) and allocates a new thread to serve it. It fetches the method body and creates the execution context. ASSIGN-LOCAL assigns a value to a local variable. If the statement to be executed is an assignment of an expression that can be reduced to a value, then the mapping of local variables is updated accordingly. NEW-ACTIVE creates a new activity that contains a new active object. It picks a fresh activity name, and assigns serialised object parameters: the initial local store of the activity is the piece of store referenced by the parameters. INVK-ACTIVE performs an

$$\textsc{Serve}\quad \frac{\mathrm{ready}(q,p,Rq) \qquad q=(f,m,\overline{v}) \qquad \mathrm{bind}(o_\alpha,m,\overline{v})=\{\ell \mid s\}}{\textsc{act}(\alpha,o_\alpha,\sigma,p,Rq :: q :: Rq') \to \textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell|s\}\}\uplus p,Rq :: Rq')}$$

$$\textsc{Assign-Local}\quad \frac{x\in\mathrm{dom}(\ell) \qquad v=[\![e]\!]_{(\sigma+\ell)}}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell \mid x=e;s\}::F\}\uplus p,Rq)\to\textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell[x\mapsto v] \mid s\}::F\}\uplus p,Rq)}$$

$$\textsc{New-Active}\quad \frac{\mathit{fields}(\mathtt{C})=\overline{x} \qquad o,\gamma \ \text{ fresh} \qquad \sigma'=\{o\mapsto[\overrightarrow{x=v}]\}\cup\mathrm{serialise}(\overline{v},\sigma) \qquad [\![\overline{e}]\!]_{(\sigma+\ell)}=\overline{v}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell \mid x=\mathtt{newActive\ C}(\overline{e});s\}::F\}\uplus p,Rq)\\ \to \textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell \mid x=\gamma;s\}::F\}\uplus p,Rq)\ \ \textsc{act}(\gamma,o,\sigma',\varnothing,\varnothing)\end{array}}$$

$$\textsc{Invk-Active}\quad \frac{[\![e]\!]_{(\sigma+\ell)}=\beta \qquad [\![\overline{e}]\!]_{(\sigma+\ell)}=\overline{v} \qquad f,o \ \text{ fresh} \qquad \sigma_1=\sigma\cup\{o\mapsto f\} \qquad (\overline{v_r},\sigma_r)=\mathrm{rename}_{\sigma'}(\overline{v},\mathrm{serialise}(\overline{v},\sigma)) \qquad \sigma''=\sigma'\cup\sigma_r}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{q\mapsto\{\ell \mid x=e.\mathtt{m}(\overline{e});s\}::F\}\uplus p,Rq)\ \textsc{act}(\beta,o_\beta,\sigma',p',Rq')\\ \to \textsc{act}(\alpha,o_\alpha,\sigma_1,\{q\mapsto\{\ell \mid x=o;s\}::F\}\uplus p,Rq)\ \ \textsc{act}(\beta,o_\beta,\sigma'',p',Rq'::(f,m,\overline{v_r}))\ \ \textsc{fut}(f,\bot)\end{array}}$$

$$\textsc{Return}\quad \frac{v=[\![e]\!]_{(\sigma+\ell)}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{(f,m,\overline{v})\mapsto\{\ell \mid \mathtt{return}\ e;s_r\}\}\uplus p,Rq)\ \textsc{fut}(f,\bot)\\ \to \textsc{act}(\alpha,o_\alpha,\sigma,p,Rq)\ \ \textsc{fut}(f,v,\mathrm{serialise}(v,\sigma))\end{array}}$$

$$\textsc{Update}\quad \frac{\sigma(o)=f \qquad (v_r,\sigma_r)=\mathrm{rename}_\sigma(v,\sigma') \qquad \sigma''=\sigma[o\mapsto v_r]\cup\sigma_r}{\textsc{act}(\alpha,o_\alpha,\sigma,p,Rq)\ \textsc{fut}(f,v,\sigma')\to\textsc{act}(\alpha,o_\alpha,\sigma'',p,Rq)\ \ \textsc{fut}(f,v,\sigma')}$$

Fig. 7: Semantics of MultiASP

asynchronous remote method invocation on an active object. It creates a fresh future with undefined value. The arguments of the invocation are serialised and put in the store of the invoked activity, possibly renaming locations to avoid clashes. The special case $\alpha = \beta$ requires a trivial adaptation of this rule (not shown here). RETURN is triggered when a request finishes. It stores the value computed by the request as a future value. Serialisation is necessary to pack the objects referenced by the future value. UPDATE updates a future reference with a resolved value. This is performed at any time when a future is referenced and the future value is resolved. Finally, the main effect of the missing rules is to modify the local store (NEW-OBJECT and ASSIGN-FIELD) and to affect the execution context (INVK-PASSIVE and RETURN-LOCAL).

Threading Policies. We extend the above semantics to specify the threading policies featured in multiactive objects (see Section 2.2). First, we extend the syntax of MultiASP so that the threading policy can be programmatically changed from a *soft limit*, i.e. a thread blocked in a wait-by-necessity is not counted in the limit, to a *hard limit*, i.e. all threads are counted in the limit:

$$s ::= \ldots \mid \textbf{setLimitSoft} \mid \textbf{setLimitHard}$$

Each request q belongs to a group $group(q)$. The filter $p\big|_g$ gives, among the active threads p, only requests of group g. There is a thread limit $\mathcal{L}_g$ defined for each group. We tag each of the currently served request as either *active* or *passive*. p contains then two kinds of served requests: active ones, noted $q_A \mapsto F$, and passive ones, noted $q_P \mapsto F$. Active(p) returns the number of active requests in p. Finally, each activity is either in a *soft limit* state written $\text{ACT}(\ldots)_S$ (by default at activity creation), or in a *hard limit* state written $\text{ACT}(\ldots)_H$. sh is a variable ranging over S and H. MultiASP semantics is modified as follows:

- Each rule allowing a thread to progress requires now that the thread is active, i.e. q is replaced by q_A in all rules except Serve and Update.
- The rule Serve is only triggered if the thread limit is not reached, i.e. if $\text{Active}(p\big|_{group(q)}) < \mathcal{L}_g$. Simmilarly, a rule for activating a thread is added:

$$\textsc{Activate-Thread}\quad \frac{Group(q) = g \qquad \text{Active}(p\big|_g) < \mathcal{L}_g}{\text{ACT}(\alpha, o_\alpha, \sigma, \{q_P \mapsto F\} \uplus p, Rq)_{sh} \to \text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto F\} \uplus p, Rq)_{sh}}$$

- There are two additional rules for switching the kind of limit, we show one hereafter (Set-Soft-Limit is the reverse):

$$\textsc{Set-Hard-Limit}\quad \text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid \mathbf{setLimitHard}; s\} :: F\} \uplus p, Rq)_{sh} \to \text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid s\} :: F\} \uplus p, Rq)_H$$

- If the kind of limit is a *soft limit*, a wait-by-necessity passivates the current thread[5]; a rule for method invocation on a future is added:

$$\textsc{Invk-Future}\quad \frac{[\![e]\!]_{(\sigma+\ell)} = o \qquad \sigma(o) = f}{\text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid x = e.\mathtt{m}(\overline{e}); s\} :: F\} \uplus p, Rq)_S \to \text{ACT}(\alpha, o_\alpha, \sigma, \{q_P \mapsto \{\ell \mid x = e.\mathtt{m}(\overline{e}); s\} :: F\} \uplus p, Rq)_S}$$

4 Example-Driven Translation Principles

In this section, we informally present the ProActive backend for ABS, that translates ABS programs into ProActive code. Basically, this section shows how the formal translation that will be defined in Section 5 is instantiated in practice in ProActive. This backend is based on the existing Java backend for ABS. We keep the translation of the functional layer unchanged and provide a translation of the object and concurrency layers.

Object Addressing and Invocation. To handle the differences between two active object languages, one needs first to define what happens when a new object (active or not) is created. As translating each ABS object into a ProActive active object is not a viable solution (because it is not scalable and because it requires a complex synchronisation of processes), we put several objects under the control of one active object, which fits the active object group model of

[5] Wait-by-necessity occurs only in case of method invocation on a future since field access is only allowed on the current object.

ABS. To this end, in the translation, we introduce a class `COG` for representing ABS COGs; only objects of the `COG` class are active objects in the ProActive translation. We translate the ABS `new` statement that creates a new object in a new COG:

```
1 Server server = new Server();
```

This instruction is translated into ProActive by the ProActive backend:

```
1 Server server = new Server();
2 COG cog = PAActiveObject.newActive(COG.class, new Object[]{Server.class}, node);
3 server.setCog(cog);
4 cog.registerObject(server);
```

Line `1` creates a regular server object. Lines `2` uses the `newActive` ProActive primitive to create a new COG active object. Additionally to the constructor parameters, ProActive allows the specification of the node onto which the active object is deployed. Line `3` makes the local server aware of its COG. Finally in line `4`, due to the ProActive by-copy parameter passing, the server object is copied in the local memory space of the newly created remote COG, and is thus locally accessible there. For objects created with `new local` in ABS, the ProActive backend simply registers them locally in the current COG. To enable the same object invocation model as in ABS, we use a two-level reference system in the ProActive translation: each COG is accessible by a global reference and each translated ABS object is accessible inside its COG through a local identifier. The pair (COG, identifier) is a unique reference for each object and allows the runtime to retrieve any object. When objects are transmitted between COG (e.g. as parameter of method invocations), a lightweight copy is transmitted by the ProActive middleware; it can be used to reach the original object by using its COG and identifier. As only the COG and the identifier are needed to reference an ABS object, we tune the object serialisation mechanism so that only those fields are transmitted between active objects, thus saving memory and bandwidth. The same strategy can be applied to translate any language featuring active object groups into non uniform active objects. For uniform active objects, creating one active object per translated object handles straightforwardly the translation but limits scalability; grouping several objects behind a same active object (proxy) would produce a more efficient program.

In order to explain now how we translate ABS asynchronous method calls in ProActive, consider the following ABS asynchronous method call:

```
1 server!start(param1, param2);
```

In ProActive such a call becomes a remote method invocation. In order to handle it with our object translation model, we perform a generic method call (implicitly asynchronous) named `execute`, on the COG of the translated `server` object:

```
1 server.getCog().execute(server.getId(), "start", new Object[]{param1, param2});
```

When run, the `execute` method of the `COG` class retrieves the target object through its identifier and runs the `start` method on it by reflection with the given parameters. Upon `execute` remote call, objects `param1` and `param2` are copied to

the memory space of the retrieved COG. Consequently, two copies of `param1` and `param2` exist in the translation whereas only one of them exists in ABS. However, if method calls occurs on them, the requests for those objects always go to the COG that manages those objects. This callback ensures that only one copy of a translated object is manipulated, like in ABS. Consequently, the behaviour by reference of ABS-like languages can be simulated with the behaviour by copy of ProActive. This mechanism is also applied for future updates.

Cooperative Scheduling. Active object languages often support sophisticated threading models and have constructs to impact on the scheduling of requests. Those constructs can be translated into adequate request scheduling of multiactive objects. For demonstration, we consider here the translation that the ProActive backend gives for ABS `await` statements (representative of cooperative scheduling), and for ABS `get` statements (representative of explicit futures).

- `await` statements on futures. An await statement on an unresolved futures releases the execution thread, for example:

```
1 await startedFut?;
```

In order to have the same behavior in the ProActive translation, we force a wait-by-necessity. We use the `getFutureValue` ProActive primitive to do that:

```
1 PAFuture.getFutureValue(startedFut);
```

As in ProActive a wait-by-necessity blocks the thread, we need to configure the ProActive COG class with multiactive object annotations (see Section 2.2) in order to qualify the `execute` method and to specify a soft thread limit:

```
1 @Group(name="scheduling", selfCompatible=true)
2 @DefineThreadConfig(threadPoolSize=1, hardLimit=false)
3 public class COG {
4   ...
5   @MemberOf("scheduling")
6   public ABSValue execute(UUID objectID, String methodName, Object[] args) {...}
7 }
```

This configuration allows a thread to process an `execute` request while a current thread that processes another `execute` request is waiting for a future. Indeed, the `hardLimit=false` parameter ensures that the threads counted in the limit (of 1 thread) are only *active* threads. In the example, the thread can be handed over to another `execute` request if `startedFut` is not resolved, just like in ABS.

- `get` statements. The ABS `get` statement blocks the execution thread to retrieve a future's value, as for example on the previous future variable:

```
1 Bool started = startedFut.get;
```

The ProActive backend translates this ABS instruction into the following code:

```
1 getCog().switchHardLimit(true); // the retrieved COG is local: the call is synchronous
2 PAFuture.getFutureValue(startedFut);
3 getCog().switchHardLimit(false);
```

This temporarily hardens the threading policy (i.e. all threads are counted in the thread limit) so that no other thread can start while the future is awaited.

- Other synchronisation constructs. We also tackled the translation of ABS `suspend` statements and of `await` statements on conditions. In this paper, we only provide the formal definition of their translation in Section 5. The details of their translation into ProActive code can be found in [17].

Wrap Up and Applicability. In order to finalise the ProActive backend for ABS, we add deployment information in the translation; for that we use the deployment descriptor embedded in ProActive: configuration files binding virtual nodes to physical machines. On the ABS side, `new cog` is followed by the name of a node for deployment. This is the *only* modification that ABS programs must incur to be executed in a distributed way. An experimental evaluation (in Appendix D) shows that a significant speedup can be achieved by a distributed execution of an ABS program thanks to the ProActive backend. It also shows that the program obtained with the ProActive backend incurs an overhead of less than 10% compared to a native ProActive application.

We have presented in details the ProActive backend for ABS and discussed the translation of common active object constructs. The concepts applied in the case of ABS are generic and can systematically turn various active object languages into deployable active objects. As an example, JCoBox is similar enough to ABS so that the approach presented here is straightforwardly applicable. The most challenging aspect is that JCoBox features a globally accessible and immutable memory, which could be translated into one active object, or which could rely on copies since the immutable property holds. Regarding Creol, in which all objects are active, the best approach is to group several objects behind a same proxy for performance reasons. Then, preserving the semantics of Creol relies on a precise interleaving of local threads. The transposition to AmbientTalk is trickier on the scheduling aspect, due to the existence of callbacks. However, we found that a callback on a future can be translated as a request that is ready to run but that starts by a wait-by-necessity on the adequate future.

5 Translational Semantics

This section formalises the translation given by the ProActive backend by introducing the translational semantics from ABS to MultiASP. We refer to Figure 1 for the concurrent object layer of ABS. Runtime syntax and semantics of ABS can be found in Appendix A. Most of the translation from ABS to MultiASP impacts statements. The rest of the source structure (classes, interfaces, methods) is unchanged except the two following:

1) We define a new class `COG`. It has methods to store and retrieve local objects, and to execute a method on a local object; `UUID` is the type of object identifiers:

```
Class COG {
  UUID freshID()
  UUID register(Object x, UUID id)
  Object retrieve(UUID id)
  Object execute(UUID id, MethodName m, params) {w=this.retrieve(id); x=w.m(params); return x}
}
```

2) All translated ABS classes are extended with two parameters: a *cog* parameter, storing the COG to which the object belongs, and an *id* parameter, storing the object's identifier in that COG; methods *cog*() and *myId*() return those two parameters; a dummy method *get*() that returns `null` is added to each object.

The translation of statements and expressions is shown in Figure 8. Each of them is explained below. *Object instantiation* first gets a fresh identifier from the current COG. Then, the new object is created with the current COG and the identifier[6]. It is stored in a reserved temporary local variable *no*. Finally, the object is referenced in the current COG and stored in x. *Object instantiation in a new* COG is similar to object instantiation in the current COG but method invocations on *newcog* variable are asynchronous remote method calls. The new object is thus copied to the memory space of the remote new COG via the *register* invocation, before being assigned to x. *Await future* uses the dummy *get()* method, that all translated objects have, in order to trigger the wait-by-necessity mechanism and potentially block the thread if the future is not resolved. *Get future* sets a hard limit on the current activity, so that no other thread starts, and then restores the soft limit after having waited for the future. *Await on conditions* performs sequential *get()* within an activity in soft limit. Conditional guards are detailed later in this section. *Asynchronous method call* retrieves the COG of the object and relies on the *execute* asynchronous method call as described in Section 4. *Synchronous local method call* distinguishes two cases, like in ABS. Either the call is local and an execution context is pushed in the stack, or the call is remote and, like in ABS, we perform an asynchronous remote method invocation and immediately wait the associated future within an activity in hard limit. Finally, instructions that do not deal with method invocation, future manipulation, or object creation, are kept unchanged.

In the translation, there exist different multiactive object groups and each group has its own thread limit. Group g_1 encapsulates $freshId$ requests; those requests cannot execute in parallel safely, so g_1 is not self compatible and can only use one thread at a time. Group g_2 gathers *execute* requests. It is limited to one thread to comply with the threading model of ABS, and the requests are self compatible to enable interleaving. Group g_3 contains *register* requests that are self compatible and that have an infinite thread limit. Concerning compatibility between groups, they are all compatible except g_3 and g_2: their compatibility is defined dynamically such that an *execute* request and a *register* request are compatible only if they do not affect the same identifier. In summary:

$$group(freshId) = g_1 \qquad group(execute) = g_2 \qquad group(register) = g_3$$
$$\mathcal{L}_{g_1} = 1 \qquad \mathcal{L}_{g_2} = 1 \qquad \mathcal{L}_{g_3} = \infty$$
$$\forall q, q'.\ (q \neq q' \neq freshId() \wedge (\nexists id.q = register(x, id) \wedge q' = execute(id, m, \overline{e}))) \Rightarrow compatible(q, q')$$

In order to support ABS conditional guards, for each guard g, we generate a method *condition_g* that takes as parameters the needed local variables $\overline{x}$. The method body can normally access the fields of the object `this`. A condition evaluation g is defined as follows: $condition\_g(\overline{x}) = \texttt{while}(\neg g)\ skip; \texttt{return null}$.

[6] The step in which the COG of the new object is set in ProActive is directly encoded in the object constructor in MultiASP.

$$[\![x = e!m(\overline{e})]\!] \triangleq t = e.cog(); id = e.myId(); \quad x = t.execute(id, m, \overline{e})$$

$$[\![await\ x?]\!] \triangleq w = x.get()$$

$$[\![await\ g \wedge g']\!] \triangleq [\![await\ g]\!]; [\![await\ g']\!]$$

$$[\![x = y.get]\!] \triangleq \textbf{setLimitHard}; w = y.get(); \textbf{setLimitSoft}; x = y$$

$$[\![x = e.m(\overline{e})]\!] \triangleq a = e.cog(); b = \texttt{this}.cog(); \texttt{if}(a == b)\ \{x = e.m(\overline{e})\} \texttt{else}\ \{t = e.cog(); id = e.myId(); x = t.execute(id, m, \overline{e}); \textbf{setLimitHard}; w = x.get(); \textbf{setLimitSoft}\}$$

$$[\![x = e]\!] \triangleq x = e$$

$$[\![x = \texttt{new local}\ C(\overline{e})]\!] \triangleq t = \texttt{this}.cog(); id = t.freshId(); no = \texttt{new C}(\overline{e}, t, id); z = t.register(no, id); x = no$$

$$[\![x = \texttt{new C}(\overline{e})]\!] \triangleq newcog = \texttt{newActive COG}(); id = newcog.freshId(); no = \texttt{new C}(\overline{e}, newcog, id); z = newcog.register(no, id); x = no$$

$$[\![await\ g]\!]_{\overline{x}} \triangleq \texttt{if}(\neg g)\ \{\ t = \texttt{this}.cog();\ id = \texttt{this}.myId();\ z = t.execute\_condition(id, condition\_g, \overline{x}); w = z.get\ \}$$

$$[\![suspend]\!] \triangleq t = \texttt{this}.cog();\ id = \texttt{this}.myId();\ z = t.execute\_condition(id, condition\_True, \overline{x}); w = z.get$$

Fig. 8: Translational semantics from ABS to MultiASP

We encode the suspend statement the same way with a *True* condition. We define an *execute_condition* method in the `COG` class; it executes generated condition methods. The *execute_condition* method has its own group with an infinite thread limit because any number of conditions can evaluate in parallel. More formally, we have: $group(execute\_condition) = g_4 \quad \mathcal{L}_{g_4} = \infty$

6 Translation Equivalence and Active Object Insights

Proving that MultiASP executions exactly simulate ABS semantics is not possible by direct bisimulation of the two semantics. Instead, we prove two different theorems stating under which conditions each semantics simulates the other. We present all technical details on the equivalence and the proof in Appendix C. We summarise below the highlights of the proof, the principles of the underlying equivalence between MultiASP and ABS terms, the differences between the languages and the restrictions of the proof.

Communication and request serving ordering. The semantics of ABS relies on a completely asynchronous communication scheme while MultiASP ensures causal ordering of requests. The equivalence can only be valid for the ABS reductions that preserve causal ordering of requests. Also, MultiASP serves requests in FIFO order, so similarly we execute a FIFO service of ABS requests, like in the existing Java backend for ABS. Note that those differences are more related to scheduling and communication patterns than to the nature of the two languages.

Shallow translation. ABS requests, COGs and futures respectively match one-to-one MultiASP requests, active objects and futures. Likewise, except for COG objects, for each ABS object there exist several copies of this object in MultiASP,

all with the same COG and the same identifier, but only one of those copies (the one hosted in the right COG) is equivalent to the ABS object.

Futures. Because of the difference between the future update mechanisms of ABS and MultiASP, the equivalence relation can follow as many local future indirections in the store as necessary. A variable holding a pointer to a future object in MultiASP is equivalent to the same variable holding directly the future reference in ABS. But also, the equivalence can follow future references in ABS: a future might have been updated transparently in MultiASP while in ABS, the explicit future read has not been performed yet.

Equating MultiASP *and ABS configurations.* A crucial part of the correctness proof consists in stating whether an ABS and a MultiASP configuration are considered equivalent. The principles of this equivalence are the following:

- Equivalence can "follow futures": A MultiASP value v is equivalent to an ABS future provided the future's value is equivalent to v; indeed in MultiASP a future can be automatically updated earlier than in the ABS case.
- Objects are identified by their identifier and their COG name: the value of the object fields are meaningless except in the COG that initially created the object. It is in this COG that we check that fields are equivalent.
- Equivalence between requests distinguishes two cases. 1) active tasks: there is a single active task per COG in ABS and it must correspond to the single active thread serving an *execute* request in MultiASP. The second element in the call stack corresponds to the invoked request. 2) inactive tasks in ABS correspond either to passive requests being currently interrupted or to not-yet-served requests in MultiASP. For each task, equivalence of executed statements, of local variables, and of corresponding future is checked.

Observational equivalence. The precise formulation of our theorems proves that the ABS behaviour is faithfully simulated by our translation (Appendix C.4) and conversely (Appendix C.5). This is proven by adequately choosing the *observable* and *not observable* actions in the weak simulation. For example remote method invocation, object creation, and field assignment can be observed and faithfully simulated. The most striking observable reduction in ABS that is not always observable in MultiASP is the future value update. For example, in ABS the configurations (a) $\mathit{fut}(f, f')$ $\mathit{fut}(f', \bot)$ and the configuration (b) $\mathit{fut}(f, \bot)$ are observationally different, whereas in MultiASP they are not. Indeed, in MultiASP, there is no process able to detect whether the first future has been updated or not. However, this example is artificial as no information is stored in the first future of configuration (a); any access to the future's value will have to follow indirections and eventually access the value that is not a future. Thus, transparency of futures and of future updates create an intrinsic difference between the two languages. This is why, in the theorem, we exclude the possibility to have a future's value being a future in the configuration. Eliminating syntactically such programs is not possible, thus we reason on reductions for which the value of a future is not a future; this is not a major restriction on expressiveness because it is still possible to have a future value that is an object containing a future (as future wrappers).

In the other direction, namely from MultiASP to ABS, the translation adds several steps in the reduction. However, the added sequences of actions never introduce concurrency so equivalence still holds because we can ignore additional local actions such as assignments and method calls that are not in the ABS program source (e.g. *myId()*).

Theorem 1 (ABS to MultiASP**).** *The translation simulates all ABS executions with FIFO policy and rendez-vous communications provided that no future value is a future reference.*

Theorem 2 (MultiASP **to ABS).** *Any reduction of the* MultiASP *translation corresponds to a valid ABS execution.*

Globally, our translational semantics fully respects the ABS semantics and simulates exactly all executions complying to the aforementioned restrictions, which either are already existing restrictions of the Java backend for ABS, or for which we have given relevant alternatives.

7 Conclusion

This paper tackled the question of providing active object languages, aimed at modelling and verification, with systematic deployment for distributed computing. For that, we have identified the necessary design choices for active object models and languages, involving: object referencing, language transparency, and request scheduling. These design choices have to be considered when implementing any active object language. We have introduced MultiASP, a multi-threaded active object language that has showed to be expressive enough to embody the main paradigms of ABS, featuring in particular cooperative scheduling. We demonstrated how to translate the constructs of an easy to program and verify active object language into the executable code of an efficient and scalable active object middleware. We have instantiated our approach by translating ABS into the ProActive middleware, that implements MultiASP in Java. The immediate outcome of this work is a ProActive backend for ABS. Our approach could be quite easily ported other active object languages since we reason more on active object abstractions than on language specifics. Typically, our work can be straightforwardly adapted to any active object language featuring cooperative scheduling, like Creol and JCoBox. Porting our results on AmbientTalk only requires minor adaptations. A comparison of the ProActive backend against a currently developed Java 8 backend for ABS [19] is ongoing. This analysis focuses on the different implementation approaches for efficiently encoding the ABS semantics. More generally, the provided proof of correctness highlighted the intrinsic differences between active object languages and models. This work will help active object users to choose the language that is the most adapted for their needs, and also help active object designers to identify the implication of specific language constructs and abstractions.

References

1. G. Agha and C. Hewitt. Concurrent programming using actors. In *Foundations of Software Technology and Theoretical Computer Science*. Springer, 1985.
2. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static analyzer for concurrent objects. In *TACAS'14*. Springer, 2014.
3. N. Bezirgiannis and F. Boer. *SOFSEM 2016*, chapter ABS: A High-Level Modeling Language for Cloud-Aware Programming, pages 433–444. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
4. F. S. D. Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP'07*. Springer, 2007.
5. S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. Johnsen, K. Pun, S. Tarifa, T. Wrigstad, and A. Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In *Formal Methods for Multicore Programming*, LNCS. Springer, 2015.
6. D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2004.
7. J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. In *ECOOP'06*. Springer, 2006.
8. C. Din, R. Bubel, and R. Hahnle. KeY-ABS: A deductive verification tool for the concurrent modelling language abs. In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25*, LNCS. Springer, 2015.
9. C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *POPL '95*. ACM, 1995.
10. E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in ABS. *Journal of Software and Systems Modeling*, 2014.
11. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, Feb. 2009.
12. L. Henrio, F. Huet, and Z. István. Multi-threaded active objects. In *COORDINATION'13*. Springer, 2013.
13. L. Henrio and J. Rochas. Declarative Scheduling for Active Objects. In *SAC'14*. ACM, 2014.
14. E. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO'12*. Springer, 2012.
15. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 2006.
16. R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*. 1996.
17. J. Rochas and L. Henrio. A ProActive Backend for ABS: from Modelling to Deployment. Research Report RR-8596, Sept. 2014.
18. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP'10*. Springer, 2010.
19. V. Serbanescu, K. Azadbakht, F. de Boer, C. Nagarajagowda, and B. Nobakht. A design pattern for optimizations in data intensive applications using ABS and JAVA 8. *Concurrency and Computation: Practice and Experience*, 2016.

Appendix

This appendix presents more details about the ProActive backend for ABS and its associated proof of correctness. More precisely, it adds:

- A recall of ABS runtime syntax and semantics, for self contents.
- The full semantics of MultiASP.
- The proofs of Theorem 3 and Theorem 4, with restrictions, equivalence relation, and preliminary lemmas.
- An experimental evaluation of the ProActive backend for ABS.

A ABS Runtime Syntax and Semantics

We first recall the runtime syntax of ABS below. ABS runtime syntax is taken from [2].

$$
\begin{array}{rll}
cn & ::= & \epsilon \mid \mathit{fut}(f, \mathit{val}) \mid \\
& & \mathit{ob}(o, a, p, q) \mid \\
& & \mathit{invoc}(o, f, \mathtt{m}, \overline{v}) \mid \\
& & \mathit{cog}(c, \mathit{act}) \mid cn\ cn \\
p & ::= & \{l \mid s\} \mid \mathtt{idle} \\
q & ::= & \epsilon \mid \{l \mid s\} \mid q\ q \\
s & ::= & \mathtt{cont}(f) \mid \ldots
\end{array}
\qquad
\begin{array}{rl}
\mathit{act} & ::= o \mid \varepsilon \\
\\
\\
\\
\mathit{val} & ::= v \mid \bot \\
a & ::= [\cdots, x \mapsto v, \cdots] \\
v & ::= o \mid f \mid \ldots
\end{array}
$$

We also recall the semantics of ABS on Figure 9 and Figure 10. The ABS semantics presented below is based on the one presented in [2], which itself represents a more mature ABS semantics than the original one in [4] (minor updates). The modifications we make on the semantics of ABS simply aim at making the proof clearer on concurrency aspects, and still keeping the essence of the translation. In particular, below is the list of the modifications we bring:

- Class fields and class parameters distinction have been removed, one of them simply being a syntactic sugar for fields initialised automatically to `null`.
- When an object is created, it is **`idle`** and has no request in queue instead of having the init request in the queue (New-Object case) or as current task (New-Cog-Object case). Consequently, a new cog coming from a new object has no current activated object instead of having the new object activated. As we enforce causal ordering of requests, it is easy to perform an init invocation on the newly created object; it will necessarily be executed first.

[2] E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in ABS. *Journal of Software and Systems Modeling*, 2014.

[4] E. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO'12*. Springer, 2012.

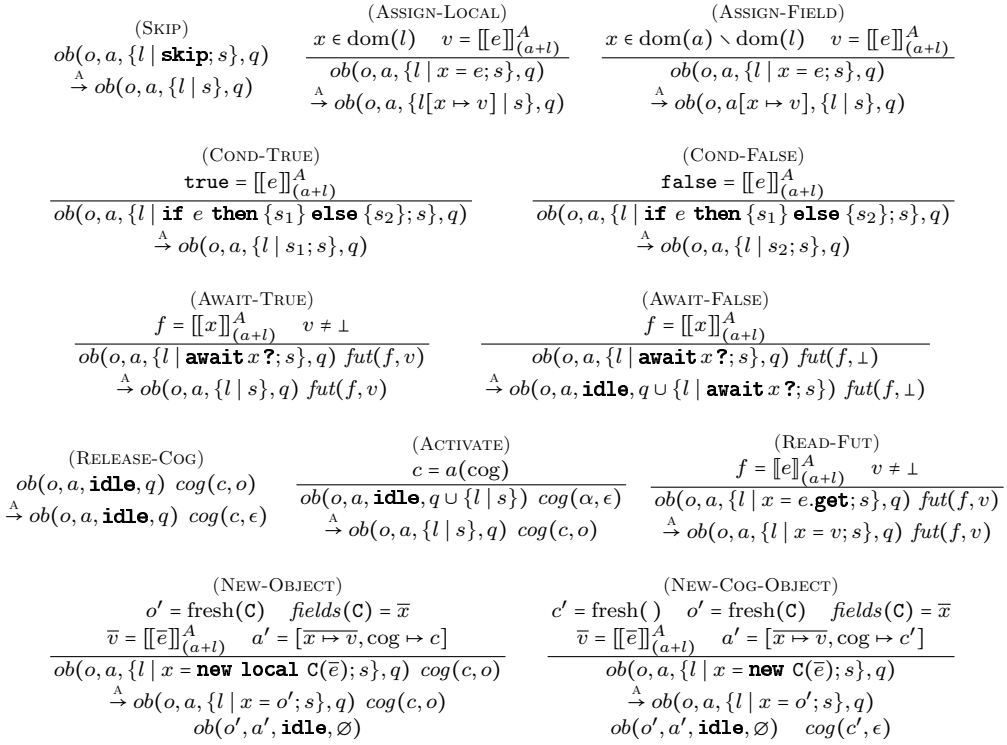


Fig. 9: Semantics of `core ABS`(1).

– Statements following a **`return`** instruction are discarded. This fits better with most of mainstream programming languages, and, in addition, having a second **`return`** in further statements would cause a deadlock in the semantics since there would be no recipient for the future and the rule cannot be reduced. Additionally, the Java backend for ABS already implement this semantics for **`return`**.

B MultiASP Semantics

Figure 11 recalls the semantics of MultiASP, with the Assign-Local, New-Object, Invk-Passive, and Return-Local rules.

C Proof of Theorem 3 and Theorem 4

In this section, we prove two theorems that corroborate the translation from ABS to MultiASP. The theorems specify under which conditions each semantics simulates the other. We first focus the proof on the less trivial and most

$$\frac{f = \text{fresh}(\,) \quad o' = [\![e]\!]^A_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \quad p'' = \text{bind}(o', f, m, \overline{v}, \texttt{class}(o'))}{ob(o, a, \{l \mid x = e\texttt{!m}(\overline{e}); s\}, q)\ ob(o', a', p', q') \xrightarrow{A} ob(o, a, \{l \mid x = f; s\}, q)\ ob(o', a', p', q' \cup p'')\ fut(f, \bot)}\ (\textsc{Rendez-vous-Comm})$$

$$\frac{cn \xrightarrow{A} cn'}{cn\ cn'' \xrightarrow{A} cn'\ cn''}\ (\textsc{Context})$$

$$\frac{o' = [\![e]\!]^A_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \quad f = \text{fresh}(\,) \quad c = a'(\text{cog}) \quad f' = l(\text{destiny}) \quad \{l' \mid s'\} = \text{bind}(o', f, \texttt{m}, \overline{v}, class(o'))}{ob(o, a, \{l \mid x = e\texttt{.m}(\overline{e}); s\}, q)\ ob(o', a', \texttt{idle}, q')\ cog(c, o) \xrightarrow{A} ob(o, a, \texttt{idle}, q \cup \{l \mid \texttt{await}\ f\texttt{?}; x = f\texttt{.get}; s\})\ fut(f, \bot)\ ob(o', a', \{l' \mid s'; \texttt{cont}\ f'\}, q')\ cog(c, o')}\ (\textsc{Cog-Sync-Call})$$

$$\frac{c = a'(\text{cog}) \quad f = l'(\text{destiny})}{ob(o, a, \{l \mid \texttt{cont}\ f\}, q)\ cog(c, o)\ ob(o', a', \texttt{idle}, q' \cup \{l' \mid s\}) \xrightarrow{A} ob(o, a, \texttt{idle}, q)\ cog(c, o')\ ob(o', a', \{l' \mid s\}, q')}\ (\textsc{Cog-Sync-Return-Sched})$$

$$\frac{f' = l(\text{destiny}) \quad o = [\![e]\!]^A_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \quad f = \text{fresh}(\,) \quad \{l' \mid s'\} = \text{bind}(o, f, \texttt{m}, \overline{v}, class(o))}{ob(o, a, \{l \mid x = e\texttt{.m}(\overline{e}); s\}, q) \xrightarrow{A} ob(o, a, \{l' \mid s'; \texttt{cont}(f')\}, q \cup \{l \mid \texttt{await}\ f\texttt{?}; x = f\texttt{.get}; s\})\ fut(f, \bot)}\ (\textsc{Self-Sync-Call})$$

$$\frac{v = [\![e]\!]^A_{(a+l)} \quad f = l(\text{destiny})}{ob(o, a, \{l \mid \texttt{return}\ e; s\}, q)\ fut(f, \bot) \xrightarrow{A} ob(o, a, \texttt{idle}, q)\ fut(f, v)}\ (\textsc{Return})$$

$$\frac{o' = [\![e]\!]^A_{(a+l)} \quad f = \text{fresh}(\,) \quad a(\text{cog}) \neq a'(\text{cog})}{ob(o, a, \{l \mid x = e\texttt{.m}(\overline{e}); s\}, q)\ ob(o', a', p, q') \xrightarrow{A} ob(o, a, \{l \mid f = e\texttt{!m}(\overline{e}); x = f\texttt{.get}; s\}, q)\ ob(o', a', p, q')}\ (\textsc{Rem-Sync-Call})$$

$$\frac{f = l'(\text{destiny})}{ob(o, a, \{l \mid \texttt{cont}(f)\}, q \cup \{l' \mid s\}) \xrightarrow{A} ob(o, a, \{l' \mid s\}, q)}\ (\textsc{Self-Sync-Return-Sched})$$

Fig. 10: Semantics of `core ABS`(2).

informative theorem: the proof that all ABS executions with FIFO policy and rendez-vous based communications are simulated by MultiASP executions. For the second proof, stating that a MultiASP translation corresponds to a valid ABS execution, we expose the differences and similarities compared to the first theorem proof and conclude.

C.1 Restrictions

Before simulating the semantics of ABS and MultiASP, we define here three specific restrictions. First, MultiASP ensures causal ordering of communications with a rendez-vous preceding all asynchronous method calls: the request *is dropped* in the remote request queue *synchronously*. This brief synchronisation does not exist in ABS where requests can arrive in any order. We will reason on $\xrightarrow{A}$, the ABS semantics with rendez-vous communications, i.e. the semantics taken from [4] where the message sending and reception rule is replaced by:

[4] E. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO'12*. Springer, 2012.

SERVE
$$\frac{\text{ready}(q,p,Rq) \qquad q=(f,m,\overline{v}) \qquad \text{bind}(o_\alpha,m,\overline{v})=\{\ell \mid s\}}{\textsc{act}(\alpha,o_\alpha,\sigma,p,Rq :: q :: Rq') \to \textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell|s\}\} \uplus p, Rq :: Rq')}$$

ASSIGN-LOCAL
$$\frac{x \in \text{dom}(\ell) \qquad v = [\![e]\!]_{(\sigma+\ell)}}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=e;s\} :: F\} \uplus p, Rq) \to \textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell[x \mapsto v] \mid s\} :: F\} \uplus p, Rq)}$$

ASSIGN-FIELD
$$\frac{\ell(this)=o \qquad x \in \text{dom}(\sigma(o)) \qquad x \notin dom(\ell) \qquad \sigma' = \sigma[o \mapsto (\sigma(o)[x \mapsto [\![e]\!]_{(\sigma+\ell)}])]}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=e;s\} :: F\} \uplus p, Rq) \to \textsc{act}(\alpha,o_\alpha,\sigma',\{q \mapsto \{\ell \mid s\} :: F\} \uplus p, Rq)}$$

NEW-OBJECT
$$\frac{\mathit{fields}(\texttt{C})=\overline{x} \qquad o \ \text{ fresh} \qquad \sigma' = \sigma \cup \{o \mapsto [\overrightarrow{x=v}]\} \qquad [\![\overline{e}]\!]_{(\sigma+\ell)}=\overline{v}}{\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=\texttt{new C}(\overline{e});s\} :: F\} \uplus p, Rq) \to \textsc{act}(\alpha,o_\alpha,\sigma',\{q \mapsto \{\ell \mid x=o;s\} :: F\} \uplus p, Rq)}$$

NEW-ACTIVE
$$\frac{\mathit{fields}(\texttt{C})=\overline{x} \qquad o,\gamma \ \text{ fresh} \qquad \sigma' = \{o \mapsto [\overrightarrow{x=v}]\} \cup \text{serialise}(\overline{v},\sigma) \qquad [\![\overline{e}]\!]_{(\sigma+\ell)}=\overline{v}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=\texttt{newActive C}(\overline{e});s\} :: F\} \uplus p, Rq) \\ \to \textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=\gamma;s\} :: F\} \uplus p, Rq) \ \ \textsc{act}(\gamma,o,\sigma',\varnothing,\varnothing)\end{array}}$$

INVK-ACTIVE
$$\frac{\begin{array}{c}[\![e]\!]_{(\sigma+\ell)}=\beta \qquad [\![\overline{e}]\!]_{(\sigma+\ell)}=\overline{v} \\ f,o \ \text{ fresh} \qquad \sigma_1=\sigma \cup \{o \mapsto f\} \qquad (\overline{v_r},\sigma_r)=\text{rename}_{\sigma'}(\overline{v},\text{serialise}(\overline{v},\sigma)) \qquad \sigma''=\sigma' \cup \sigma_r\end{array}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=e.\texttt{m}(\overline{e});s\} :: F\} \uplus p, Rq) \ \ \textsc{act}(\beta,o_\beta,\sigma',p',Rq') \\ \to \textsc{act}(\alpha,o_\alpha,\sigma_1,\{q \mapsto \{\ell \mid x=o;s\} :: F\} \uplus p, Rq) \ \ \textsc{act}(\beta,o_\beta,\sigma'',p',Rq' :: (f,m,\overline{v_r})) \ \ \textsc{fut}(f,\bot)\end{array}}$$

INVK-PASSIVE
$$\frac{[\![e]\!]_{(\sigma+\ell)}=o \qquad [\![\overline{e}]\!]_{(\sigma+\ell)}=\overline{v} \qquad \text{bind}(o,m,\overline{v})=\{\ell' \mid s'\}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid x=e.\texttt{m}(\overline{e});s\} :: F\} \uplus p, Rq) \\ \to \textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell' \mid s'\} :: \{\ell \mid x=\bullet;s\} :: F\} \uplus p, Rq)\end{array}}$$

RETURN-LOCAL
$$\frac{v=[\![e]\!]_{(\sigma+\ell)}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell \mid \textbf{return}\ e;s_r\} :: \{\ell' \mid x=\bullet;s\} :: F\} \uplus p, Rq) \\ \to \textsc{act}(\alpha,o_\alpha,\sigma,\{q \mapsto \{\ell' \mid x=v;s\} :: F\} \uplus p, Rq)\end{array}}$$

RETURN
$$\frac{v=[\![e]\!]_{(\sigma+\ell)}}{\begin{array}{c}\textsc{act}(\alpha,o_\alpha,\sigma,\{(f,m,\overline{v}) \mapsto \{\ell \mid \textbf{return}\ e;s_r\} \uplus p, Rq) \ \ \textsc{fut}(f,\bot) \\ \to \textsc{act}(\alpha,o_\alpha,\sigma,p,Rq) \ \ \textsc{fut}(f,v,\text{serialise}(v,\sigma))\end{array}}$$

UPDATE
$$\frac{\sigma(o)=f \qquad (v_r,\sigma_r)=\text{rename}_\sigma(v,\sigma') \qquad \sigma''=\sigma[o \mapsto v_r] \cup \sigma_r}{\textsc{act}(\alpha,o_\alpha,\sigma,p,Rq) \ \ \textsc{fut}(f,v,\sigma') \to \textsc{act}(\alpha,o_\alpha,\sigma'',p,Rq) \ \ \textsc{fut}(f,v,\sigma')}$$

Fig. 11: Semantics of MultiASP

$$
\begin{array}{l}
1\ \exists a.\, cog(\alpha,a) \in Cn \ \underline{\text{iff}}\ \ \exists o_\alpha,\sigma,p,Rq.\, act(\alpha,o_\alpha,\sigma,p,Rq) \in \underline{Cn} \ \ \underline{\text{with}} \\
2\ \ \exists \overline{v},p',q.ob(i\_\alpha,\overrightarrow{x\mapsto v},p',q) \in Cn \ \underline{\text{iff}}\ \ \exists o,\overline{v'}.\,\sigma(o)=[cog\mapsto\alpha, myId\mapsto i,\overrightarrow{x\mapsto v'}]\ \underline{\text{with}} \\
3\ \ \ \overline{v} \approx_\sigma^{Cn} \overline{v'} \ \ \wedge \\
4\ \ \ \left(\begin{array}{l}\exists l,s.\,p'=\{l|s\}\ \underline{\text{iff}}\ \ \exists f,i,m,\overline{v''},\ell',s',\ell'',s''.\,\big((f,execute,i,m,\overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \\ \wedge\ \ell'(\texttt{this})=o\big)\ \ \underline{\text{with}}\ \ \forall x \in dom(l)\backslash destiny.\, l(x) \approx_\sigma^{Cn} \ell'(x) \wedge l(destiny)=f) \wedge s \approx_{\sigma+\ell'}^{Cn} s'\end{array}\right) \ \wedge \\
5\ \ \ \ \forall f.\left(\begin{array}{l}\big(\exists l,s.(\{l|s\} \in q \wedge l(destiny)=f)\ \underline{\text{iff}} \\ \exists i,m,\overline{v''},\ell',s',\ell'',s''.\big(\big((f,execute,i,m,\overline{v''})_P \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge \ell'(\texttt{this})=o\big) \\ \vee\ \big((f,execute,i,m,\overline{v''}) \in Rq \wedge o_\alpha.retrieve(i)=o \wedge \text{bind}(o,m,\overline{v''})=\{\ell'|s'\}\big)\big) \\ \ \ \underline{\text{with}}\ \ \ (\forall x \in dom(l)\backslash destiny.\, l(x) \approx_\sigma^{Cn} \ell'(x)) \wedge s \approx_{\sigma+\ell'}^{Cn} s'\big)\end{array}\right) \\
6\ fut(f,v) \in Cn \ \underline{\text{iff}}\ \ \exists v',\sigma.\,(\textsc{fut}(f,v',\sigma) \in \underline{Cn} \wedge \text{Method}(f)=execute)\ \underline{\text{with}}\ \ v \approx_\sigma^{Cn} v' \\
7\ fut(f,\bot) \in Cn \ \underline{\text{iff}}\ \ \textsc{fut}(f,\bot) \in \underline{Cn} \wedge \text{Method}(f)=execute
\end{array}
$$

Fig. 12: Equivalence condition of two configurations[7,8]

$$
\textsc{Rendez-vous-Comm}\ \frac{\text{fresh}(f) \qquad i'\_\beta = [\![e]\!]^A_{a+l} \qquad \overline{v} = [\![\overline{e}]\!]^A_{a+l} \qquad p'' = \text{bind}(i'\_\beta, f, m, \overline{v}, \texttt{class}(o'))}{\begin{array}{c} ob(i\_\alpha,a,\{l|x=e!m(\overline{e});s\},q)\ ob(i'\_\beta,a',p',q') \\ \xrightarrow{\text{A}} ob(i\_\alpha,a,\{l|x=f;s\},q)\ ob(i'\_\beta,a',p',q' \cup p'')\ fut(f,\bot)\end{array}}
$$

where $[\![]\!]^A_\sigma$ is the expression evaluation in ABS. Another rule has to be added in case of a message to the same object. Secondly, in MultiASP, while thread activation can happen in any order, the order in which requests are served is FIFO by default instead of the non-deterministic activation of a thread featured by ABS semantics. In both the existing Java backend for ABS and the developed ProActive backend, activation and request service are FIFO, although multi-active objects support the definition of different policies [3]. Consequently, we only reason on executions with a *FIFO policy*, i.e. executions that serve requests and restore them in a FIFO order. Finally, the proof does not deal with local synchronous method invocations; indeed this would make the proofs more entangled and would bring no particular insight on the translation because this part is similar to the existing Java backend for ABS.

C.2 Equivalence

We define an equivalence relation $\mathcal{R}$ between MultiASP and ABS terms. We then prove that any single step of one calculus can be simulated by a sequence of steps

[7] We use the following notation: $\exists y.\, P \ \underline{\text{iff}}\ \exists x.\, Q \ \underline{\text{with}}\ R$ means $(\exists y.\, P \ \underline{\text{iff}}\ \exists x.\, Q) \wedge \forall x,y.\, P \wedge Q \Rightarrow R$. This allows R to refer to x and y.

[8] Method(f) returns the method of the request corresponding to future f

[3] L. Henrio and J. Rochas. Declarative Scheduling for Active Objects. In *SAC'14*. ACM, 2014.

in the other. This is similar to the proof in [1]; we use the same observation notion: processes are observed based on remote method invocations.

Definition 1 (Equivalence Relation $\mathcal{R}$). *$\approx_\sigma^{Cn}$ is an equivalence between values (or between a value and a storable) in the context of a* MultiASP *store σ and an ABS configuration Cn:*

$$v \approx_\sigma^{Cn} v \qquad f \approx_\sigma^{Cn} f \qquad i\_\alpha \approx_\sigma^{Cn} [cog \mapsto \alpha, Id \mapsto i, \overline{x \mapsto v'}]$$

$$\frac{v \approx_\sigma \sigma(o)}{v \approx_\sigma^{Cn} o} \qquad \frac{fut(f, v') \in Cn \qquad v' \approx_\sigma^{Cn} v}{f \approx_\sigma^{Cn} v}$$

Runtime values in ABS are either (global) object references, future references or primitive values. Equivalence $\approx_\sigma^{Cn}$ specifies that two identical values or futures are equivalent if they are the same. Otherwise, an object is characterised by its identifier and its COG name. The two last cases are more interesting and are necessary because of the difference between the future update mechanisms of ABS and MultiASP. First, the equivalence can follow as many local indirections in the store as necessary. Second, the equivalence can also follow future references in ABS: a future might have been updated transparently in MultiASP while in ABS the explicit future read has not been performed yet.

Furthermore, we have an equivalence on statements: $s \approx_{(\sigma+\ell')}^{Cn} s'$ iff

- *either $[\![s]\!] = s'$*
- *or $s = (x = v; s_1) \ \wedge \ s' = (x = e; [\![s_1]\!])$ with $v \approx_\sigma^{Cn} [\![e]\!]_{(\sigma+\ell')}$*

In words, two statements are equivalent if one is the translation of the other. Or, two statements are also equivalent if both statements start with an assignment of equivalent values to the same variable, followed by equivalent statements.

Finally, ABS configuration Cn and MultiASP *configuration $\underline{Cn}$ are equivalent, written $Cn \ \mathcal{R} \ \underline{Cn}$, iff the condition on Figure 12 holds.*

The equivalence condition of Figure 12 considers three cases:

– The first five lines deal with equivalence of COG. This case compares both activity content and activity requests on the ABS and MultiASP sides:

- To compare activity content, we rely on the fact that activities have the same name in ABS and in MultiASP. Each ABS object *ob* must correspond to one equivalent MultiASP object in the equivalent activity α. The object equivalent to *ob* must be in the store, must reference α as its COG, must have i as identifier[9], and must have other fields equivalent to the ones of *ob*.
- To compare activity requests, we need to compare the tasks that exist in ABS *ob* terms to the tasks that exist in the corresponding MultiASP *act* terms. We consider again two cases:
  1. Either the task is active, the single active task of *ob* (in p') must have exactly one equivalent active task in MultiASP (in p). In MultiASP, this

[9] Recall that the corresponding ABS identifier is i_α.

[1] M. Dam and K. Palmskog. Location independent routing in process network overlays. In *PDP'14*. IEEE, 2014.

task must have two elements in the current stack[10]: the call to the COG (the *execute* call) and the redirected call to the targeted object o, where o is equivalent to ob. In addition, values of local variables must be equivalent, except *destiny* that must correspond to the future of the MultiASP request. Finally, the current thread of the two tasks must be equivalent according to the equivalence on statement.

2. Otherwise the task is inactive, and two cases are possible. Either the task has already started and has been interrupted: in this case the comparison is similar to the active task comparison above. Or the task has not started yet: there must be a corresponding task in the request queue Rq of the MultiASP active object α, and we check that (i) the future is equivalent, (ii) the invoked object o is equivalent to ob, and (iii) the method body retrieved by the bind predicate is equivalent to the ABS task.

– Line 6 deals with equivalence of resolved futures. A future's value in MultiASP refers to its local store. Two resolved futures are equivalent if their values are equivalent. In MultiASP, only futures from *execute* method calls are considered.
– Line 7 deals with equivalence of unresolved futures. In that case, both futures must be unresolved to ensure equivalence.

C.3 Preliminary lemmas

In order to establish a proper link between ABS and MultiASP semantics, we first identify activity names of MultiASP with COG names (ranged over by α, β). Object locations are valid only locally in MultiASP and globally in ABS, their equivalent global reference is the pair (activity name, *Id* fields). We suppose that each object name in ABS is of the form i_α where α is the name of the COG where the object is created, and i is unique locally in α. The semantics then allows us to choose *Id* and i so that they are equal. By convention, Cn ranges over ABS configurations and $\underline{Cn}$ over MultiASP ones.

Secondly, we have the following property which says that in an ABS configuration, if an object ob has an active request that is not `idle`, then there exists a COG in which ob is the current active object:

Lemma 1.

$$ob(i\_\alpha, \overrightarrow{x \mapsto v}, p, q) \in Cn \wedge p \neq \texttt{idle} \Rightarrow cog(\alpha, i\_\alpha) \in Cn$$

Proof. By induction on the ABS reduction rules. □

We also rely on the following lemmas for equivalence of ABS and MultiASP configurations. The first lemma below mentions equivalence of values (Lemma 2). The next lemma is a prerequisite for assessing equivalence of expression evaluation (Lemma 3). The last lemma relates equivalence of values with serialization and renaming; these aspects are indeed essential in a distributed context (Lemma 4).

[10] Recall that we only deal with asynchronous method calls in the proof.

Lemma 2 (Equivalence of values).

$$v \approx_\sigma^{Cn} v' \Rightarrow v \approx_\sigma^{Cn} [\![v']\!]$$

Proof. We prove this lemma by recurrence on the proof of $v \approx_\sigma^{Cn} v'$ (Definition 1 of $\mathcal{R}$). This corresponds to a recurrence on the number of indirections of references that are followed to evaluate v'. If v is a primitive value then $v' = v$ and the equivalence $v \approx_\sigma^{Cn} v$ is direct because $v = [\![v]\!]$. Otherwise[11], the rule $\dfrac{v \approx_\sigma^{Cn} \sigma(o)}{v \approx_\sigma^{Cn} o}$ is used, $v' = o$, and $v \approx_\sigma^{Cn} \sigma(o)$, and we need to handle three cases:

- $\sigma(o) = f$. In this case, since $[\![o]\!] = o$ by the evaluation function of Figure 6, then $v \approx_\sigma^{Cn} o$.
- $\sigma(o) = [\overrightarrow{x \mapsto v}]$. In this case, the evaluation of $[x \mapsto v]$ is equal to $[x \mapsto v]$ ($[\![[x \mapsto v]]\!] = [x \mapsto v]$), so we also have $v \approx_\sigma^{Cn} o = [\![o]\!]$.
- Else $\sigma(o) = v''$. In this case, since $v \approx_\sigma^{Cn} v''$, we have $v \approx_\sigma^{Cn} [\![v'']\!]$ by recurrence hypothesis. Then $v \approx_\sigma^{Cn} [\![o]\!] = [\![\sigma(o)]\!] = [\![v'']\!]$ (by the evaluation function of Figure 6).

Lemma 3 (Equivalence of Eval functions). *Let Cn be an ABS configuration and suppose $Cn\ \mathcal{R}\ \underline{Cn}$.*

Let $ob(o\_\alpha, a, \{l|s\}, q) \in Cn$. By definition of $\mathcal{R}$, there exists a single activity $\text{ACT}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$, with $\sigma(o) = [cog \mapsto \alpha, myId \mapsto i, a']$ and $(f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge \ell'(\texttt{this}) = o$.

For any ABS expression e we have:

$$[\![e]\!]^A_{a+l} \approx_\sigma^{Cn} [\![e]\!]_{\sigma+\ell'}$$

Proof. We only consider here the case where x is a variable, because the other cases are not different from the original translation from ABS to Java (not different from the original Java backend). In particular, the evaluation of arithmetic expressions relies on the equivalence of variables and the fact that the evaluation of the arithmetic expression itself is the same.

We prove the equivalence of evaluation functions on variables by case analysis.

- Either the variable belongs to the fields of the object $ob(o\_\alpha, a, \{l|s\}, q)$, and not to the local variables: $a(x) = v \wedge v \notin dom(l)$. In this case we have $ob(o\_\alpha, a, \{l|s\}, q) \in Cn$ with $x \mapsto v \in a$ and, by definition of equivalence, $\text{ACT}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$, $\exists (f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} \in p$ with, $\forall x \in dom(l) \backslash destiny$, $l(x) = v \approx_\sigma^{Cn} \ell'(x) = v'$ by line 8 of equivalence condition, and $\sigma(o) = [mycog \mapsto \alpha, myId \mapsto i, a']$ by line 2. Also by definition of equivalence, $a'(x) = v'$ and $v \approx_\sigma^{Cn} v'$ (because σ is the store of α). We have $[\![x]\!]_{\sigma+\ell'} = [\![v']\!]$ by the definition of the evaluation function of Figure 6, with $\ell'(this) = o$ by the equivalence condition of figure 12 (because $x \notin dom(l)$ and hence, $x \notin dom(\ell')$). Thus we have $v \approx_\sigma^{Cn} [\![x]\!]_{\sigma+\ell'} = [\![v']\!]_{\sigma+\ell'}$ by Lemma 2.

[11] The other cases are not applicable here, e.g. a future f is not a valid MultiASP value.

– Or the variable belongs to local variables: $l(x) = v$. In this case we have $ob(o\_\alpha, a, \{l|s\}, q) \in Cn$ with $l(x) = v$ and, by definition of equivalence, $\text{ACT}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$, $\exists(f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} \in p$ with, $\forall x \in dom(l) \backslash destiny$, $l(x) = v \approx_\sigma^{Cn} \ell'(x) = v'$ by line 8 of equivalence condition. Finally, $[\![x]\!]_{a+l}^A \approx_\sigma^{Cn} \ell'(x)$ and by Lemma 2 $[\![x]\!]_{a+l}^A \approx_\sigma^{Cn} [\![\ell'(x)]\!]_{\sigma+\ell'}$.

Also, serialization, together with the related renaming of local references, are crucial points of difference between ABS and MultiASP; we need the following lemma to deal with this aspect.

Lemma 4 (Serialization and equivalence).

$$v \approx_\sigma^{Cn} v' \wedge \sigma' = serialise(\sigma, v') \Rightarrow v \approx_\sigma^{Cn} v'$$
$$\overline{v} \approx_\sigma^{Cn} \overline{v'} \wedge (\overline{v''}, \sigma') = rename_\sigma(\overline{v'}, \sigma) \Rightarrow \overline{v} \approx_{\sigma'}^{Cn} \overline{v''}$$

Proof. The proof relies mostly on the definition of equivalence $\mathcal{R}$. Concerning serialization, the main difference is the fact that we check the equivalence in a smaller (or equal) store, however this store is self-contained and every reference that could be followed by the equivalence still exist in the serialized store. Concerning renaming, the proof is trivial as only the case where referenced are followed is changed and the renaming ensures the equivalence of the reached values.

Finally, to deal with the proof of Theorem 3 we rely on the fact that all ABS objects are locally registered in their COG:
Invariant Reg. For every activity α such that o_α is the location of the active object of activity α in its store σ_α. If the current task consists in an invocation to $o_\alpha.retrieve(i)$, then this invocation succeeds, because the object has been registered first; the invocation returns some object o' such that $i\_\alpha \approx_\sigma^{Cn} o'$.

Proof. To prove the **Invariant Reg** we will be relying on the translational semantics provided in Figure 8. Suppose that a call to $o_\alpha.retrieve(i)$ is performed; we want to show that this call succeeds and returns an object equivalent to i_α. If we have such a call, it means that we are in the body of an $execute(id, m, \overline{parameters})$ call, with $id = i$. Since the *execute* method is reserved, we know that we ran a remote method call, corresponding to an expression $e.m(\overline{parameters})$ with $e.cog() = \alpha$ and $e.myId() = id$. Also, we have e such that $[\![e]\!]_{\sigma_\beta+\ell} = o$ and $\sigma_\beta(o) = [cog \mapsto \alpha, myId \mapsto id, \cdots]$.

Additionally, we know that we cannot have method calls on the temporary variable *no*, that is used to temporarily store a newly created object until it is registered to its COG, because this variable is only known in the instantiation thread and no method call is performed on it. Thus, since e cannot be *no*, it is necessarily a variable x that has been assigned after a call to $t.register(no)$, where $t \mapsto \alpha$, that makes the COG t aware of the new object referenced by *no* (we also use the fact that a pointer could not be forged).

Consequently, the call to *register* precedes the remote call $e.m(\overline{parameters})$. Now, we have two cases.

- Either the *register* call is local because it comes along with a new local object instantiation (a call to **`new local`**). In this case, since the *register* call is synchronous, it is necessarily finished before $e.m(\overline{parameters})$ is invoked, so the later $o_\alpha.retrieve(i)$ succeeds naturally.
- Or the *register* call is remote because it comes along with a new remote object instantiation (a call to **`new`**). In this case, since the *register* call is asynchronous, it may happen that the *register* and the *execute* requests are both awaiting in the queue. However, thanks to the rendez-vous communication ordering and the FIFO service policy, we know that the *register* request is in queue before the *execute* request, and the *register* request will be served first. Now, considering the multiactive compatibilities defined for the groups of *register* and *execute*, we see that an incompatibility is raised between the two requests because they access the same identifier. Therefore, the $o_\alpha.retrieve(i)$ request cannot be served before the *register* request finishes, and thus, the $o_\alpha.retrieve(i)$ of the later *execute* request succeeds.

In both cases one can notice that $i\_\alpha \approx_\sigma^{Cn} o'$.

C.4 Core Proof of Theorem 3 - From ABS to MultiASP

We first aim at proving that MultiASP semantics simulates any ABS execution. Let $P = \overline{IC}\ \{\ \overline{x\ ;}\ s\ \}$ be an ABS program, let Cn_0 be the initial ABS configuration corresponding to this program: $ob(start, \varnothing, p, \varnothing)$, where the process p corresponds to the activation of the program's main block: $p = \{l|s\}$. The initial MultiASP configuration corresponding to program $[\![P]\!]$ is $\textsc{act}(\alpha_0, o, \sigma_0[o \mapsto \varnothing], q_0 \mapsto \{l|[\![s]\!]\}, \varnothing)$. It is easy to see that this initial configuration has the same behaviour as $\textsc{act}(\alpha_0, o, \sigma_0[o \mapsto \varnothing, start \mapsto \varnothing],\ (f, execute, start, m) \mapsto \{l|[\![s]\!]\} :: \{\varnothing | x = \bullet\}, \varnothing)$. We denote this new MultiASP initial configuration $\underline{Cn_0}$, and $\rightarrow^*$ denotes the transitive closure of $\rightarrow$.

Theorem 3. *The translation simulates all ABS executions with FIFO policy and rendez-vous communications:*
$Cn_0 \overset{\text{A}}{\rightarrow}{}^* Cn$ *with a FIFO policy* $\wedge \nexists f, f'.\ fut(f, f') \in Cn$
$\Rightarrow\ \exists \underline{Cn}.\ \underline{Cn_0} \rightarrow^* \underline{Cn}\ \wedge\ Cn\ \mathcal{R}\ \underline{Cn}$

Because of the simulation technique, what matters is to decide which actions are observable. Preceding works use the sending of requests as an observable. In our case, we show in the proof that request sending is simulated exactly, as well as method return, object creation, and field assignment. Assignment to local variables is trickier because of the existence of temporary variables created by the translation. Consequently, assignments to ABS local variables can be observed exactly, but it is not the case for assignments to MultiASP local variables. The most striking example of an observable reduction in ABS that is not observable in MultiASP is the case of future's value update. Indeed, transparency of futures and of future updates create an intrinsic difference between the two languages. This is why, in the theorem, we exclude the possibility to have a future's value being a future in the configuration. In MultiASP, there is no process able to detect whether the first future has been updated or not: in ABS the configurations (a)

$fut(f, f')$ $fut(f', \bot)$ and the configuration (b) $fut(f, \bot)$ are observationally different, whereas in MultiASP they are not. However, this example is purely artificial as no information is stored in the first future of configuration (a). Indeed, any access the value stored in a future will have to follow indirections and eventually access the future f'. Eliminating syntactically such programs is not possible, thus we only reason on reductions for which the value of a future is not a future; this is not a major restriction on expressiveness because it is always possible to have a future value that is an object containing a future. The identification of observability differences in active object languages is a major result: it gives a significant insight on the design of programming languages. Besides, we could make this observation because we chose a faithful translation that matches one to one as much as possible requests, objects and futures of ABS to the equivalent notions in MultiASP.

Proof. Theorem 3 is proved by induction on the ABS reduction. It relies on the definition of equivalence $\mathcal{R}$ on Figure 12 and on the following induction step:
If $Cn_0 \xrightarrow{A}{}^{*} Cn$, $Cn \,\mathcal{R}\, \underline{Cn}$, $Cn \xrightarrow{A} Cn'$ with a FIFO policy, and
$\nexists f, f'.\, fut(f, f') \in Cn'$, then $\exists \underline{Cn'}.\ \underline{Cn} \rightarrow^{*} \underline{Cn'} \ \wedge\ Cn' \,\mathcal{R}\, \underline{Cn'}$.

The proof is done by case analysis on the ABS reduction rules applied, each of which is detailed hereafter. First note that, in the definition of equivalence between values and between statements ($\approx_\sigma^{Cn}$), the ABS configuration Cn is only used to track futures, and thus in all the cases except RETURN, $\approx_\sigma^{Cn}$ and $\approx_\sigma^{Cn'}$ are the same. Each case is concluded by argumnents on observability. We are interested in observing remote method invocations, return of asynchronous calls, assignments to field variables, assignment to local variables that were not created by the translation into MultiASP. Other reductions are considered to be non-observable and will be used transparently and can be as many times as necessary and to simulate any ABS reduction.

• **Case of [ASSIGN-LOCAL] rule**
We first consider ASSIGN-LOCAL rule, to show that the translation of this rule through the ProActive backend is correct, i.e. we can observe equivalent configurations on ABS and MultiASP side after reduction. In this rule, a particular ABS configuration Cn leads to a configuration Cn', noted $Cn \xrightarrow{A} Cn'$, as follows:

$$\frac{x \in \mathrm{dom}(l) \quad v = [\![e]\!]^{A}_{(a+l)}}{ob(i\_\alpha, a, \{l \mid x = e; s\}, q) \xrightarrow{A} ob(i\_\alpha, a, \{l[x \mapsto v] \mid s\}, q)}$$

Suppose that Cn is also related to a MultiASP configuration $\underline{Cn}$ by the definition of equivalence, noted $Cn \,\mathcal{R}\, \underline{Cn}$. Then, we want to show that a configuration $\underline{Cn'}$, equivalent to Cn', can be obtained by MultiASP semantics from $\underline{Cn}$. In the case of ASSIGN-LOCAL rule, it means that a local variable x must exist in $\underline{Cn'}$ and that its value after assignment must be equivalent to the value assigned to it in ABS.

To begin with, by Lemma 1 there must be a COG in the ABS starting configuration where i_α is the current active object: $\exists cog(\alpha, i\_\alpha) \in Cn$. Then, by definition of the equivalence relation $\mathcal{R}$, we have the following key points:

- First, there exists a corresponding MultiASP activity in the MultiASP starting configuration:
  $\exists o_\alpha, \sigma, p, Rq.act(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$.
- Second, this activity has an active request initiated from an *execute* method call, and maps to a current statement to execute:
  $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''.((f, execute, i, m, \overline{v''})_A \mapsto$
  $\{\ell'|s'\} :: \{\ell''|s''\} \in p$ with $(x = e; s) \approx_\sigma^{Cn} s'$
  and $\forall x \in dom(l) \setminus destiny.l(x) \approx_\sigma^{Cn} \ell(x)$.
  Thus, by the definition of equivalence between statements, either $s' = (x = e; [\![s]\!])$, we denote $e' = e$; or $e = v$ (because ABS statement is $x = v; s$) and $s' = (x = e'; [\![s]\!])$ with $v \approx_\sigma^{Cn} [\![e']\!]_{\sigma+\ell'}$.

Those key points allow us to find the starting MultiASP configuration $\underline{Cn}$ and to apply the MultiASP ASSIGN-LOCAL reduction rule:

$$\frac{x \in \mathrm{dom}(\ell') \qquad v' = [\![e']\!]_{(\sigma+\ell')}}{\text{ACT}(\alpha, o_\alpha, \sigma, p, Rq) \rightarrow \qquad \text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell'[x \mapsto v'] \mid [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p', Rq)}$$

We have now to verify that the obtained MultiASP configuration Cn' is in relation $\mathcal{R}$ with the ABS configuration $\underline{Cn'}$.

First, we must have at least as many corresponding terms in MultiASP as in ABS. And indeed, we have $cog(\alpha, i\_\alpha) \in Cn'$ on one hand, and $\text{ACT}(\alpha, o_\alpha, \sigma, p'', Rq) \in \underline{Cn'}$ on the other hand. We only consider the terms that changed in the reduction, not all terms of the configurations.

Second, we must check equivalence of each element by checking the equivalence of their content. We have on the ABS side the term $ob(i\_\alpha, a, \{l[x \mapsto v]|s\}, q)$ and on the MultiASP side the activity α which has in particular $q_A \mapsto \{\ell'[x \mapsto v']|[\![s]\!]\} :: \{\ell''|s''\}$ in the method execution stack, with $v' = [\![e']\!]_{\sigma+\ell'}$. Then, to have $v \approx_\sigma^{Cn'} v'$ we must have $v \approx_\sigma^{Cn'} [\![e']\!]_{\sigma+\ell'}$. Now, we have two cases:

1. either $e' = e$ by definition and therefore, by Lemma 3: $[\![e]\!]_{a+l}^A \approx_\sigma^{Cn'} [\![e]\!]_{\sigma+\ell'}$.
2. or $v \approx_\sigma^{Cn'} [\![e]\!]_{\sigma+\ell'}$ and $[\![e']\!]_{\sigma+\ell'} = v'$.

The rest of the elements of the activity are unchanged except the remaining statements, but in this case we have $s \approx_{\sigma+\ell'}^{Cn'} [\![s]\!]$ by definition of equivalence.

• Case of [ASSIGN-FIELD] rule

Like the previous case, suppose $Cn \xrightarrow{\text{A}} Cn'$ with the ASSIGN-FIELD rule:

$$\frac{x \in \mathrm{dom}(a) \setminus \mathrm{dom}(l) \quad v = [\![e]\!]_{(a+l)}^A}{ob(i\_\alpha, a, \{l \mid x = e; s\}, q) \xrightarrow{\text{A}} ob(i\_\alpha, a[x \mapsto v], \{l \mid s\}, q)}$$

With the same strategy as the previous case, namely by having first $Cn \mathcal{R} \underline{Cn}$, then $\underline{Cn} \rightarrow \underline{Cn'}$ and finally $Cn' \mathcal{R} \underline{Cn'}$, we want to show that an object field x exists in $\underline{Cn'}$ and that its value after assignment is equivalent to the value assigned to it in ABS.
First, we have again by Lemma 1, $\exists cog(\alpha, i\_\alpha) \in Cn$.
Second, by definition of the equivalence relation $\mathcal{R}$, we have:

- A corresponding MultiASP activity:
  $\exists o_\alpha, \sigma, p, Rq.act(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$.
- Values in the store σ that are equivalent to the values contained in a:
  $\exists o, \overline{v'}.\sigma(o) = [cog \mapsto \alpha, myId \mapsto i, \overrightarrow{x \mapsto v'}]$ where $\overline{v} \approx_\sigma^{Cn} \overline{v'}$ and $\overrightarrow{x \mapsto v} = a$.
  Consequently $x \in dom(\sigma(o))$ because $x \in dom(a)$.
- A statement currently executed that belongs to the current active request of the activity:
  $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''$ such that:
  $(f, execute, i, m, \overline{v''})_A \mapsto \{\ell' | s'\} :: \{\ell'' | s''\} \in p$ and $\ell'(\texttt{this}) = o$ with $(x = e; s) \approx_\sigma^{Cn} s'$.
  As the previous case here, either $s' = (x = e; [\![s]\!])$ and $e' = e$, or $e = v$ and $s' = (x = e'; [\![s]\!])$ with $v \approx_\sigma^{Cn} [\![e']\!]_{\sigma + \ell'}$.

Then, this leads to the following MultiASP ASSIGN-FIELD rule:

$$\frac{\ell'(this) = o \qquad x \in \mathrm{dom}(\sigma(o)) \qquad x \notin dom(\ell') \qquad \sigma' = \sigma[o \mapsto (\sigma(o)[x \mapsto [\![e']\!]_{(\sigma+\ell')}])]}{\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \rightarrow \qquad \textsc{act}(\alpha, o_\alpha, \sigma', \{q_A \mapsto \{\ell' \mid [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p', Rq)}$$

As in the previous case, we must now check that $Cn' \mathcal{R} \underline{Cn'}$. First, we have $cog(\alpha, i\_\alpha) \in Cn'$ and $\textsc{act}(\alpha, o_\alpha, \sigma', p'', Rq) \in \underline{Cn'}$ which are equivalent terms. Second, we have to compare the content of $ob(i\_\alpha, a[x \mapsto v], \{l \mid s\}, q)$ on one hand, and of activity α on the other hand with $\sigma(o) = [cog \mapsto \alpha, myId \mapsto i, \overrightarrow{x \mapsto v}[x \mapsto [\![e']\!]_{\sigma+\ell'}]]$. Then, we should have $\overline{v} \approx_\sigma^{Cn'} \overline{v'}$, which is unchanged except for the particular value v that must be equivalent to the evaluation of e'. Then, similarly to the ASSIGN-LOCAL case, we have again $v \approx_\sigma^{Cn'} [\![e']\!]_{\sigma+\ell'}$. Finally, we have $s \approx_{\sigma+\ell'}^{Cn'} [\![s]\!]$ by the definition of equivalence $\mathcal{R}$.

• **Case of [AWAIT-TRUE] rule**

We start from the ABS AWAIT-TRUE reduction rule, in which $Cn \xrightarrow{\mathrm{A}} Cn'$:

$$\frac{f = [\![x]\!]^A_{(a+l)} \qquad v \neq \bot}{ob(i\_\alpha, a, \{l \mid \texttt{await}\, x\, \texttt{?}; s\}, q)\ fut(f, v) \xrightarrow{\mathrm{A}} ob(i\_\alpha, a, \{l \mid s\}, q)\ fut(f, v)}$$

From the configuration Cn, we know that, by Lemma 1, there exists α such that $cog(\alpha, i\_\alpha) \in Cn$. We also know, by equivalence relation $\mathcal{R}$ and by translational semantics, that there are o_α, σ, p, Rq such that $\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$. Besides, there exist ℓ', s', ℓ'', s'' such that, by translational semantics: $\{q_A \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid s''\}\} \in p$. Also, we know that there are v', σ' such that $\textsc{fut}(f, v', \sigma') \in \underline{Cn}$, with $v \approx_{\sigma'}^{Cn} v'$. Thus, we have the following MultiASP configuration $\underline{Cn}$ such that $Cn \mathcal{R} \underline{Cn}$:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \texttt{s}''\}\} \uplus p', Rq)$$
$$\textsc{fut}(f, v', \sigma')$$

By Lemma 3, we have $[\![x]\!]^A_{(a+l)} \approx_\sigma^{Cn} [\![x]\!]_{(\sigma+\ell')}$, and, by case analysis on the definition of $\approx_\sigma^{Cn}$ (Definition 1), we have $[\![x]\!]_{(\sigma+\ell')} = o$. Then, two cases are possible in MultiASP: either $\sigma(o) = f$ (4th case and 2nd case of the definition of $\approx_\sigma^{Cn}$), or $\sigma(o) = v_f$ where $v \approx_\sigma^{Cn} v_f$ (4th case and 5th case of the definition of $\approx_\sigma^{Cn}$).

– In the first case, where o points to a future, we first perform a future update on the MultiASP configuration $\underline{Cn}$ (the point is to catch up with the ABS execution). Thus, we have $\underline{Cn} \rightarrow \underline{Cn'}$ where in $\underline{Cn'}$ activity α is:
$\text{ACT}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \mathtt{s}''\}\} \uplus p', Rq)$
with $\sigma'' = \sigma[o \mapsto v_r] \cup \sigma_r$ and $(v_r, \sigma_r) = \text{rename}_\sigma(v', \sigma')$. After that, the current local method call, equals to $f.get()$, can be performed. Consequently, after local invocation and local assignment, we have a MultiASP configuration $\underline{Cn''}$, such that $\underline{Cn'} \rightarrow^* \underline{Cn''}$, as follows:

$$\begin{array}{l}\text{ACT}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell'[w \mapsto v_r] \mid [\![s]\!]\} :: \{\ell'' \mid \mathtt{s}''\}\} \uplus p', Rq)\\ \text{FUT}(f, v', \sigma')\end{array}$$

Now, we must compare Cn' and $\underline{Cn''}$ to prove that they are equivalent. The only change concerns the value pointed to by the variable x. Suppose x is a field of the current object (the case where x is a local variable is similar). To prove the line 3 of the definition of equivalence (Figure 12), we must ensure that $f \approx^{Cn'}_{\sigma''} v_r$. Indeed, by definition of $\approx^{Cn'}_{\sigma''}$, we need to have $\sigma(o) = v_r$ and $v \approx^{Cn'}_{\sigma''} v_r$, which is true because first, $v \approx^{Cn'}_{\sigma'} v'$ and second, because by Lemma 4 we have $v \approx^{Cn'}_{\sigma'} v' \ \wedge \ (v_r, \sigma_r) = rename_{\sigma'}(v', \sigma') \Rightarrow v \approx^{Cn'}_{\sigma_r} v_r$ with $\sigma_r \subseteq \sigma''$.
Concerning activity α, the elements to be considered are local variables and statements; the other elements did not change. MultiASP configuration $\underline{Cn'}$ contains an additional local variable w that comes from the *get* invocation; this local variable is not used. The other local variables did not change, which preserves equivalence. Finally, the remaining statements in ABS and in MultiASP are guaranteed to be equivalent by the fact that $s \approx^{Cn'}_{\sigma''+\ell'} [\![s]\!]$, according to the definition of equivalence $\mathcal{R}$.
– In the second case, where o points to a value, the future has already been updated in the past. Consequently, no preliminary future update is necessary and the last steps of the previous case can be performed in the same way, but directly with $\underline{\text{Cn}}$ instead of $\underline{\text{Cn'}}$ and with v_f instead of v_r. The same arguments in favour of equivalence of Cn' and $\underline{Cn'}$ can be applied.

• **Case of [AWAIT-FALSE] rule**

We start from the ABS AWAIT-FALSE reduction rule, where $Cn \xrightarrow{\text{A}} Cn'$:

$$\frac{f = [\![x]\!]^A_{(a+l)}}{ob(i\_\alpha, a, \{l \mid \mathtt{await}\ x\ \mathtt{?}; s\}, q)\ fut(f, \bot) \xrightarrow{\text{A}} ob(i\_\alpha, a, \mathtt{idle}, q \cup \{l \mid \mathtt{await}\ x\ \mathtt{?}; s\})\ fut(f, \bot)}$$

By Lemma 1, we know that there exists α such that $cog(\alpha, i\_\alpha) \in Cn$. By the definition of equivalence $\mathcal{R}$, we also know that there exist o_α, σ, p, Rq such that $\text{ACT}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$ and that we have $\text{FUT}(f, \bot) \in \underline{Cn}$. In addition, we have by translational semantics: $\exists \ell', \ell'', s''.\{q_A \mapsto \{\ell' \mid w = [\![x]\!].get(); [\![s]\!]\} :: \{\ell'' \mid s''\}\} \in p$. Thus, we have the following MultiASP configuration $\underline{Cn}$, such that $Cn\ \mathcal{R}\ \underline{Cn}$:

$$\begin{array}{l}\text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \mathtt{s}''\}\} \uplus p', Rq)\\ \text{FUT}(f, \bot)\end{array}$$

By Lemma 3, we have $[\![x]\!]^A_{(a+l)} \approx^{Cn}_{\sigma} [\![x]\!]_{(\sigma+\ell')}$. By case analysis on the definition of $\approx^{Cn}_{\sigma}$, we have $[\![x]\!]_{(\sigma+\ell')} = o$. Then, only one case is possible in MultiASP: $\sigma(o) = f$ according to the 4th and 2nd case of the definition of $\approx^{Cn}_{\sigma}$ (no other case is possible because the future has not been resolved yet). Thus, the current statement of activity α is in fact a local method call to $x.get()$ where $\sigma(o) = f$. Since f has not been resolved yet in MultiASP, the current request of activity α switches to passive mode q_P by rule INVK-FUTURE (recall that the objects are in soft-limit by default):

$$\frac{[\![x]\!]_{(\sigma+\ell)} = o \qquad \sigma(o) = f}{\begin{array}{l}\text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \mathtt{s}''\}\} \uplus p', Rq)_S \\ \rightarrow \text{ACT}(\alpha, o_\alpha, \sigma, \{q_P \mapsto \{\ell \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \mathtt{s}''\}\} \uplus p', Rq)_S\end{array}}$$

Thus we have the following MultiASP configuration $\underline{Cn'}$, such that $\underline{Cn} \rightarrow \underline{Cn'}$:

$$\begin{array}{l} act(\alpha, o_\alpha, \sigma, \{q_P \mapsto \{\ell' \mid w = x.get(); [\![s]\!]\} :: \{\ell'' \mid \mathtt{s}''\}\} \uplus p', Rq) \\ \text{FUT}(f, \bot)\end{array}$$

Now we have to prove $Cn' \mathcal{R} \underline{Cn'}$. In MultiASP, the only element that changed in activity α is the status of the current request, from q_A to q_P which corresponds to the fact that the current task became **idle** in ABS. Indeed, in the definition of equivalence Figure 12, as soon as there is a term $\{l \mid s\}$ in p there must be a corresponding active thread in MultiASP (line 4). Similarly, passive threads in MultiASP correspond to some of the requests in the queue in ABS. The passivation of the thread correspond exactly to the transfer of the task to the request queue in ABS. The exact equivalence of the two tasks is then trivial (because $\mathtt{await}\, x; s \approx^{Cn'}_{\sigma} w = x.get(); [\![s]\!]$).

• **Case of [RELEASE-COG] rule**

We have the following ABS reduction $Cn \xrightarrow{\text{A}} Cn'$ with the RELEASE-COG rule:

$$ob(i\_\alpha, a, \mathtt{idle}, q)\ cog(\alpha, i\_\alpha) \xrightarrow{\text{A}} ob(i\_\alpha, a, \mathtt{idle}, q)\ cog(\alpha, \epsilon)$$

From configuration Cn, by definition of equivalence $\mathcal{R}$, we know that there exist o_α, σ, p, Rq such that $\text{ACT}(\alpha, o_\alpha, \sigma, p, Rq)$ is in a corresponding MultiASP configuration $\underline{Cn}$, where $Cn \mathcal{R} \underline{Cn}$. Since in Cn the current task is **idle** and all other objects in COG α are **idle** too (we know that by Lemma 1), there are no $f, i, m, \overline{v''}, \ell', s', \ell'', s''$ such that $(f, execute, i, m, \overline{v''})_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid s''\} \in p$, i.e. there is no active *execute* request in the set p of executed requests of activity α. In this case, we have $Cn' \mathcal{R} \underline{Cn}$ as the COG is the only term that changed from Cn to Cn' and the state of COGs is not relevant in the definition of equivalence.

• **Case of [ACTIVATE] rule**

We have the following ABS reduction $Cn \xrightarrow{\text{A}} Cn'$ with the ACTIVATE rule:

$$\frac{\alpha = a(\text{cog})}{ob(i\_\alpha, a, \mathtt{idle}, q \cup \{l \mid s\})\ cog(\alpha, \epsilon) \xrightarrow{\text{A}} ob(i\_\alpha, a, \{l \mid s\}, q)\ cog(\alpha, i\_\alpha)}$$

From configuration Cn, by definition of equivalence $\mathcal{R}$, we know that there exist o_α, σ, p, Rq such that $\text{ACT}(\alpha, o_\alpha, \sigma, p, Rq)$ is in a corresponding MultiASP configuration $\underline{Cn}$, where $Cn\ \mathcal{R}\ \underline{Cn}$. We also know that there is no active *execute* request in the set p of executed requests of activity α, because in the ABS configuration Cn the current task is `idle`. As the $\{l \mid s\}$ request belongs to the pending requests of the ABS object i_α, we have, by definition of equivalence $\mathcal{R}$ (Figure 12, line 5, passive requests case), two possible cases (for the two sides of the disjunction of line 5):

- Either a corresponding passive request in MultiASP is in the set of executed requests: $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''.(f, execute, i, m, \overline{v''})_P \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \ \wedge\ \ell'(\texttt{this}) = o$. The requests involved in this case can only be *execute* requests. Additionally, we have $group(execute) = g_2$ and $\mathcal{L}_{g_2} = 1$ (see end of Section 5), which basically means that only one *execute* request can be active at a time. Thus, as no other request is active in p, we have the condition: $\text{Active}(p\big|_{g_2}) < \mathcal{L}_{g_2}$ that is verified. Consequently, we can perform an ACTIVATE-THREAD MultiASP reduction to fall in configuration $\underline{Cn'}$, where $\underline{Cn} \to \underline{Cn'}$:

$$\frac{Group(q') = g_2 \qquad \text{Active}(p\big|_{g_2}) < \mathcal{L}_{g_2}}{\begin{array}{l}\text{ACT}(\alpha, o_\alpha, \sigma, \{q'_P \mapsto \{\ell'|s'\} :: \{\ell''|s''\}\} \uplus p'', Rq)_S \\ \to \text{ACT}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\}\} \uplus p'', Rq)_S\end{array}}$$

  with $q' = (f, execute, i, m, \overline{v''})$.
  We now have to prove that $Cn'\ \mathcal{R}\ \underline{Cn'}$. The only alteration takes place in the set of executed requests: there is one less (*execute*) request that is passive (one less request to be matched in line 5 of $\mathcal{R}$ in Figure 12) but one more (*execute*) request that is active: we must compare it with the current task in ABS in line 4 of $\mathcal{R}$ (Figure 12). Proving the condition of line 4 from the first case of the disjunction of line 5 is trivial: all the elements perfectly match.
- Or a corresponding request on the MultiASP side is in the queue of activity α: $\exists \ell', s', i, m, .\ (f, execute, i, m, \overline{v''}) \in Rq \wedge o_\alpha.retrieve(i) = o \wedge \text{bind}(o, m, \overline{v''}) = \{\ell'|s'\}$
  $\underline{\text{with}}\quad (\forall x \in dom(l) \backslash destiny.\, l(x) \approx_\sigma^{Cn} \ell'(x)) \wedge s \approx_{\sigma+\ell'}^{Cn} s'\big)$
  Let $q' = (f, execute, i, m, \overline{v''})$ and $Rq = Rq_0 :: q' :: Rq_1$. In this case, as no other request is active and as *execute* requests are compatible with all other requests (they are declared to safely run in parallel), this request is ready to be served because the predicate $\text{ready}(q', p, Rq_0)$ is true. Thus, we can apply the MultiASP SERVE reduction rule[12] to fall in configuration $\underline{Cn'}$, where $\underline{Cn} \to \underline{Cn'}$:

$$\frac{\text{ready}(q', p, Rq_0) \quad q' = (f, execute, i, m, \overline{v''}) \quad \text{bind}(o_\alpha, execute, \overline{v''}) = \{\ell'' \mid s''\}}{\text{ACT}(\alpha, o_\alpha, \sigma, p, Rq_0 :: q' :: Rq_1) \to \text{ACT}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell'' \mid s''\}\} \uplus p'', Rq_0 :: Rq_1)}$$

  where s'' is the body of the execute method:
  $s'' = (w = this.retrieve(i); x = w.\mathtt{m}(\overline{params}); return\ x)$.
  Furthermore, we also have $o_\alpha.retrieve(i) = o$ and thus, after a local method

[12] Here, we use the fact that we restricted ABS to the FIFO activation of requests: if a request in Rq_0 could be served it would be served first.

call to *retrieve* and a local assignment, we can perform the main local method call `m` to object o. We can then obtain a configuration $\underline{Cn''}$, where $\underline{Cn'} \rightarrow^* \underline{Cn''}$ and the MultiASP Invk-Passive reduction rule is the last step:

$$\frac{[\![w]\!]_{(\sigma+\ell'')} = o \qquad [\![\overline{params}]\!]_{(\sigma+\ell'')} = \overline{v''} \qquad \text{bind}(o, m, \overline{v''}) = \{\ell' \mid s'\}}{\begin{array}{l}\text{ACT}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell'' \mid x = w.\mathtt{m}(\overline{params}); return\ x\}\} \uplus p'', Rq) \\ \rightarrow \text{ACT}(\alpha, o_\alpha, \sigma, \{q'_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid x = \bullet; return\ x\}\} \uplus p'', Rq)\end{array}}$$

Finally, in this case also, we can prove that $Cn' \ \mathcal{R} \ \underline{Cn''}$. Indeed, there is one less request to match in the second case of the disjunction of line 5 of $\mathcal{R}$, but the new one is to be compared in line 4 of $\mathcal{R}$ (Figure 12):

$$\begin{array}{l}\exists l, s.\, p' = \{l|s\} \ \underline{\text{iff}} \ \exists f, i, m, \overline{v''}, \ell', s', \ell'', s''.\, \big((f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \\ \{\ell''|s''\} \in p \wedge \ \ell'(\mathtt{this}) = o\big) \\ \quad \underline{\text{with}} \ \forall x \in dom(l) \backslash destiny.\, l(x) \approx^{Cn}_{\sigma} \ell'(x) \wedge l(destiny) = f) \wedge s \approx^{Cn}_{\sigma+\ell'} s'\end{array}$$

Indeed, we constructed a request matching those conditions: each condition is either a direct consequence of the applied reduction rules or was ensured by the initial conditions on the request in queue. Besides, note that by definition of bind, $\text{bind}(o, m, \overline{v''}) = \{\ell' \mid s'\}$ implies $\ell'(this) = o$, which confirms the last condition above.

- **Case of [Read-Fut] rule**

We have the following ABS reduction $Cn \xrightarrow{\text{A}} Cn'$ with the Read-Fut rule:

$$\frac{f = [\![e]\!]^A_{(a+l)} \qquad v \neq \bot}{ob(i\_\alpha, a, \{l \mid x = e.\mathbf{get}; s\}, q)\ fut(f, v) \xrightarrow{\text{A}} ob(i\_\alpha, a, \{l \mid x = v; s\}, q)\ fut(f, v)}$$

To begin with, from the ABS configuration Cn, we know that, by Lemma 1, there exists α such that $cog(\alpha, i\_\alpha) \in Cn$. By the definition of equivalence $\mathcal{R}$, there are o_α, σ, p', Rq such that $\text{ACT}(\alpha, o_\alpha, \sigma, p', Rq) \in \underline{Cn}$. There also exist ℓ', s', ℓ'', s'' such that $\{q_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid s''\}\} \in p'$. By definition of equivalence on statements, we have $(x = e.\mathbf{get}; s) \approx^{Cn}_{\sigma} s'$. By translational semantics, we have $s' = [\![x = e.\mathbf{get}; s]\!]$, so we have $s' = (\mathbf{setLimitHard}; w = e.get(); \mathbf{setLimitSoft}; x = e); [\![s]\!])$. Thus, we have the following MultiASP configuration $\underline{Cn}$ where $Cn \ \mathcal{R} \ \underline{Cn}$:

$$\begin{array}{l}\text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell | \mathbf{setLimitHard}; w = e.get(); \\ \qquad\qquad \mathbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \\ \text{FUT}(f, v', \sigma')\end{array}$$

with $v \approx^{Cn}_{\sigma'} v'$ by definition of equivalence $\mathcal{R}$, and where $[\![e]\!]^A_{a+l} \approx^{Cn}_{\sigma} [\![e]\!]_{\sigma+\ell}$ by Lemma 3. Now, we can reduce configuration $\underline{Cn}$ to consume the **setLimitHard** statement with the MultiASP Set-Hard-Limit rule, and fall in configuration $\underline{Cn'}$, where $\underline{Cn} \rightarrow \underline{Cn'}$, as follows:

$$\begin{array}{l}\text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid \mathbf{setLimitHard}; w = e.get(); \mathbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq)_S \\ \quad \rightarrow \text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid w = e.get(); \mathbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq)_H\end{array}$$

Now, we have $[\![e]\!]^A_{a+l} = f$ and $[\![e]\!]^A_{a+l} \approx^{Cn}_{\sigma} [\![e]\!]_{\sigma+\ell}$, so two cases are possible:

– Either the future has not been updated yet, and $[\![e]\!]_{\sigma+\ell} = o$ and $\sigma(o) = f$. Then, a future update occurs through the MultiASP UPDATE reduction rule, and we fall in configuration $\underline{Cn''}$, where $\underline{Cn'} \to \underline{Cn''}$, as follows:

$$\frac{\sigma(o) = f \qquad (v_r, \sigma_r) = \text{rename}_\sigma(v', \sigma') \qquad \sigma'' = \sigma[o \mapsto v_r] \cup \sigma_r}{\text{ACT}(\alpha, o_\alpha, \sigma, p', Rq)\ \text{FUT}(f, v', \sigma') \to \text{ACT}(\alpha, o_\alpha, \sigma'', p', Rq)\ \text{FUT}(f, v', \sigma')}$$

where $p' = \{\ell | w = e.get(); \textbf{setLimitSoft}; x = e; [\![s]\!]\} :: \{\ell'' \mid s''\} \uplus p$. At this point, the $get()$ method call on object o can be performed and succeeds because the future has been updated. Additionally, thanks to Lemma 4, $v \approx^{Cn}_{\sigma'} v'$ implies $v \approx^{Cn}_{\sigma''} v_r$: the value of the future after serialization is preserved. Moreover, we can note that v is not a future[13], thus neither v' nor v_r can point to a future. Finally, after applying some local reduction rules: local method invocation, local assignment (changing ℓ to ℓ') and a limit switch, we end up in the following MultiASP configuration $\underline{Cn_3}$, where $\underline{Cn''} \to^* \underline{Cn_3}$:

$$\text{ACT}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell' | x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq)_S \quad \text{FUT}(f, v', \sigma')$$

From here, we can prove that $Cn' \mathcal{R} \underline{Cn_3}$. Indeed, we have $v \approx^{Cn'}_{\sigma''} v'$ and $(x = v; s) \approx^{Cn'}_{\sigma''} (x = e; [\![s]\!])$ because $[\![e]\!]_{\sigma''+\ell'} = v_r$ and $v \approx^{Cn'}_{\sigma''} v_r$. Finally, the other elements need not being considered because they did not change.

– Or the future has already been updated, and $[\![e]\!]_{\sigma+\ell} = v'$. By Lemma 3, we have $f \approx^{Cn}_{\sigma} v'$. Then, performing get on e has no visible effect, and after applying several local MultiASP reduction rules, we end up in a configuration $\underline{Cn''}$, where $\underline{Cn'} \to \underline{Cn''}$, as follows:

$$\text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' | x = e; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \quad \text{FUT}(f, v', \sigma')$$

This final configuration is similar to one of the case above. The only difference is that no future update was needed and thus the local store is unchanged. Otherwise, we can prove $Cn' \mathcal{R} \underline{Cn''}$ similarly to the case above.

• Case of [NEW-OBJECT] rule

We start from the NEW-OBJECT rule in which $Cn \xrightarrow{A} Cn'$ as follows:

$$\frac{i'\_\alpha = \text{fresh}(\texttt{C}) \quad \mathit{fields}(\texttt{C}) = \overline{x} \qquad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \quad a' = [\overline{x \mapsto v}, \text{cog} \mapsto \alpha]}{\begin{array}{c} ob(i\_\alpha, a, \{l \mid x = \texttt{new local C}(\overline{e}); s\}, q)\ cog(\alpha, i\_\alpha) \\ \xrightarrow{A} ob(i\_\alpha, a, \{l \mid x = i'\_\alpha; s\}, q)\ cog(\alpha, i\_\alpha) ob(i'\_\alpha, a', \texttt{idle}, \varnothing) \end{array}}$$

By definition of equivalence $\mathcal{R}$, there is a MultiASP activity in $\underline{Cn}$ corresponding to $cog(\alpha, i\_\alpha)$: $\exists o_\alpha, \sigma, p', Rq.\text{ACT}(\alpha, o_\alpha, \sigma, p', Rq)$.

By translational semantics, p' contains the statements that create an equivalent new object in MultiASP:
$p' = \{q_A \mapsto \{\ell \mid t = this.cog(); id = t.freshId(); no = \texttt{new C}(\overline{e}, t, id); z = t.register(no, id); x = no; [\![s]\!]\} :: E \uplus p''\}$

By definition of equivalence $\mathcal{R}$, there is a MultiASP object in $\underline{Cn}$ corresponding to $ob(i\_\alpha, a, \cdots)$:

[13] Recall that we excluded the particular execution where the value of a future is a future

$\exists o, \overline{v'}.\sigma(o) = [cog \mapsto \alpha, myId \mapsto i, \overrightarrow{x \mapsto v'}]$,
and this object maps to the current local variable this: $\ell(this) = o$.

So we have the following MultiASP configuration $\underline{Cn}$ such that $Cn\mathcal{R}\underline{Cn}$:

$$\begin{array}{l}\text{ACT}(\alpha, o_\alpha, \sigma[o \mapsto [cog \mapsto \alpha, myId \mapsto i, \overrightarrow{x \mapsto v'}]],\\ \{q_A \mapsto \{\ell[this \mapsto o] \mid t = this.cog(); id = t.freshId();\\ no = \texttt{new C}(\overline{e}, t, id); z = t.register(no, id); x = no; [\![s]\!]\} :: E \uplus p''\}, Rq)\end{array}$$

Then, $\underline{Cn}$ can be reduced by evaluating all local reductions of α until a configuration $\underline{Cn'}$ has the object creation as current statement; these reduction steps only execute local assignment and local (synchronous) method calls fetching the COG and the identifier of the invoked object. In particular, we aim at $\underline{Cn} \rightarrow^* \underline{Cn'}$ where $\underline{Cn'}$ contains the activity α with the active thread executing the object creation: $q_A \mapsto \{\ell' \mid no = \texttt{new C}(\overline{e}, t, id); z = t.register(no, id); x = no; [\![s]\!]\}$, with $\ell' = \ell[t \mapsto \alpha, id \mapsto i']$[14].

This configuration $\underline{Cn'}$ can be straightforwardly reduced to a configuration $\underline{Cn''}$, such that $\underline{Cn'} \rightarrow \underline{Cn''}$ by the MultiASP NEW-OBJECT rule as follows:

$$\frac{\begin{array}{c}fields(\texttt{C}) = \overline{x}\\ o' \quad \text{fresh} \qquad \sigma' = \sigma \cup \{o' \mapsto [cog \mapsto \alpha, myId \mapsto i', \overrightarrow{x = v''}]\} \qquad [\![\overline{e}]\!]_{(\sigma + \ell')} = \overline{v''}\end{array}}{\begin{array}{c}\underline{Cn'} \rightarrow \text{ACT}(\alpha, o_\alpha, \sigma', \{q_A \mapsto \{\ell' \mid no = o';\\ z = t.register(no, id); x = no; [\![s]\!]\} :: E\} \uplus p'', Rq)\end{array}}$$

The configuration $\underline{Cn''}$ has now the activity α that contains the new object. By now, this object has a proper identifier and points to the right activity α. The register instruction is then evaluated but we do not have to reason on it here as we already showed that the **Invariant Reg** is verified.

This leads us to the final configuration $\underline{Cn'''}$ such that $\underline{Cn''} \rightarrow^* \underline{Cn'''}$, which contains the term:

$$\text{ACT}(\alpha, o_\alpha, \sigma', \{q_A \mapsto \{\ell'' \mid x = no; [\![s]\!]\} :: E\} \uplus p'', Rq)$$

where $\ell'' = \ell'[no \mapsto o']$.

Now we can compare Cn' and $\underline{Cn'''}$ to see if equivalence of the two configurations can be met. Typically, we have two objects of the same COG affected in ABS and one activity affected in MultiASP. Overall two objects are involved, the currently active object (i_α) and the newly created one (i'_α).

First consider i'_α. We have a fresh object i'_α in ABS that must be equivalent to the fresh object o' in the store of α in MultiASP. By Definition 1 of $\mathcal{R}$, line 2, already additional fields pointing to the COG α and to the identifier i' correspond to the ABS identifier $i'\alpha$. Equivalence of other parameters of the

[14] Note that i', the same identifier as the fresh id allocated by ABS, can be chosen as a fresh identifier by the method *freshId*. This is due to the definition of equivalence between objects: if i' was not fresh in α, i'_α would already be an existing ABS object and could not be chosen as a fresh ABS object identifier.

fresh objects is obtained by Lemma 3. The newly created object is **idle** with no pending request in ABS; line 4 and 5 of Definition 1 are trivially verified.

Second, consider i_α. Only the currently served request has changed and only line 4 of Definition 1 has to be checked. The set of local variables did not change except in MultiASP where the set of local variables contains more variables. Finally, we have to check equivalence of the remaining statements. We fall in the second case of the definition of statement equivalence:
$s_{\text{ABS}} = (x = i'\_\alpha; s) \ \wedge$
$s_{\text{MultiASP}} = (x = no; [\![s]\!])$ *with* $i'\_\alpha \approx_{\sigma}^{Cn'} [\![no]\!]_{(\sigma'+\ell'')}$.
The current statement in ABS and in MultiASP indeed fits with the requirement above. Additionally, $[\![no]\!]_{(\sigma'+\ell'')} = o'$ and we have showed before equivalence of i'_α and o'.

- **Case of [NEW-COG-OBJECT] rule**

We start from the NEW-COG-OBJECT rule where $Cn \xrightarrow{\text{A}} Cn'$ as follows:

$$\frac{\beta = \text{fresh}() \quad i'\_\beta = \text{fresh}(\texttt{C}) \quad \mathit{fields}(\texttt{C}) = \overline{x} \qquad \overline{v} = [\![\overline{e}]\!]^{A}_{(a+l)} \quad a' = [\overline{x \mapsto v}, \text{cog} \mapsto \beta]}{\begin{array}{c} ob(i\_\alpha, a, \{l \mid x = \texttt{new C}(\overline{e}); s\}, q) \\ \xrightarrow{\text{A}} ob(i\_\alpha, a, \{l \mid x = i'\_\beta; s\}, q) ob(i'\_\beta, a', \texttt{idle}, \varnothing) \quad cog(\beta, \epsilon) \end{array}}$$

From the ABS configuration Cn, we know that:
- By Lemma 1, $\exists \alpha.\, cog(\alpha, i\_\alpha) \in Cn$.
- By definition of equivalence $\mathcal{R}$,
  - $\exists o_\alpha, \sigma, p', Rq.\text{ACT}(\alpha, o_\alpha, \sigma, p', Rq) \in \underline{Cn}$
  - $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''.(f, execute, i, m, \overline{v''})_A \mapsto \{\ell' \mid s'\} :: \{\ell'' \mid s''\} \in p'$ and $(x = \texttt{new C}(\overline{e}); s) \approx_{\sigma+\ell'}^{Cn} s'$.
- By definition of the equivalence on statements and by translational semantics, we have $(x = \texttt{new C}(\overline{e}); s) \approx_{\sigma+\ell'}^{Cn} s'$ implies $s' = [\![x = \texttt{new C}(\overline{e}); s]\!]$ and thus
  $s' = (newcog = newActive\ \text{COG}();$
  $id = newcog.freshId(); no = \texttt{new C}'(\overline{e}, newcog, id);$
  $z = newcog.register(no, id); x = no; [\![s]\!])$.

With all those elements, we have the following MultiASP configuration $\underline{Cn}$ such that $Cn\ \mathcal{R}\ \underline{Cn}$:

$$\begin{array}{l} \text{ACT}(\alpha, o_\alpha, \sigma, \{\texttt{q}_A \mapsto \{\ell' \mid newcog = newActive\ \text{COG}(); \\ id = newcog.freshId(); no = \texttt{new C}'(\overline{e}, newcog, id); \\ z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \end{array}$$

We can apply NEW-ACTIVE MultiASP reduction rule to fall into configuration $\underline{Cn'}$, such that[15] $\underline{Cn} \rightarrow \underline{Cn'}$:

$$\frac{\mathit{fields}(\text{COG}) = \overline{x_{\text{COG}}}}{\begin{array}{c} \beta, o_\beta \ \text{ fresh} \qquad \sigma' = \{o_\beta \mapsto [\overline{x_{\text{COG}} = v_{\text{COG}}}]\} \cup \mathit{serialise}(\overline{v_{\text{COG}}}, \sigma) \qquad [\![\overline{e_{\text{COG}}}]\!]_{(\sigma+\ell')} = \overline{v_{\text{COG}}} \\ \hline \begin{array}{c} \underline{Cn} \rightarrow \text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid newcog = \gamma; \\ id = newcog.freshId(); no = \texttt{new C}'(\overline{e}, newcog, id); \\ z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \qquad \text{ACT}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \end{array} \end{array}}$$

[15] Note that we can pick β as a fresh activity name because, by definition of $\mathcal{R}$, as β is free in ABS , it is also free in MultiASP

Now, we can reduce $\underline{Cn'}$ to a configuration $\underline{Cn''}$ by two local assignments and a remote invocation (to get a fresh identifier), until the current statement is the new object creation:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell''' \mid no = \texttt{new C}'(\overline{e}, newcog, id); z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \quad \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \quad \textsc{fut}(f, \bot)$$

where $\ell''' = \ell'[newcog \mapsto \beta, id \mapsto f]$.

Then, it is possible to reduce the configuration until a fresh identifier is found and returned, and the future has been updated in α: $\underline{Cn''} \to^* \underline{Cn_3}$ such that the following elements are in the configuration[16]:

$$\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell''' \mid no = \texttt{new C}'(\overline{e}, newcog, id); z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \quad \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \quad \textsc{fut}(f, i', \varnothing)$$

with $\ell'''(id) = i'$.

We can apply the New-Object MultiASP reduction rule to end up in a configuration $\underline{Cn_4}$ such that $\underline{Cn_3} \to \underline{Cn_4}$ by applying the following reduction:

$$\frac{fields(\texttt{C}) = \overline{x} \qquad o \text{ fresh} \qquad \sigma'' = \sigma \cup \{o \mapsto [cog \mapsto \beta, myId \mapsto f, \overrightarrow{x = v''}]\} \qquad [\![\overline{e}]\!]_{(\sigma + \ell')} = \overline{v''}}{\begin{array}{c}\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell''' \mid no = \texttt{new C}'(\overline{e}, newcog, id); \\ z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \\ \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \quad \textsc{fut}(f, i', \varnothing) \to \textsc{act}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell''' \mid no = o; \\ z = newcog.register(no, id); x = no; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p'', Rq) \\ \textsc{act}(\beta, o_\beta, \sigma', \varnothing, \varnothing) \quad \textsc{fut}(f, i', \varnothing)\end{array}}$$

Then, the final steps consist in reducing the local assignment and evaluating the register remote method invocation. This brings us to the final configuration $\underline{Cn_5}$, where $\underline{Cn_4} \to^* \underline{Cn_5}$, which contains the following terms:

$$\textsc{act}(\alpha, o_\alpha, \sigma'', \{q_A \mapsto \{\ell_4 \mid x = no; [\![s]\!]\} :: E\} \uplus p'', Rq)$$

$$\textsc{act}(\beta, o_\beta, \sigma_3, \varnothing, (f', register, o', i')) \quad \textsc{fut}(f, i', \varnothing) \quad \textsc{fut}(f', \bot)$$

with $\ell_4 = \ell'''[no \mapsto o]$ and $\sigma_3(o') = [cog \mapsto \beta, myId \mapsto i', \overline{x \mapsto v_2}]$ is the serialisation of o.

Now, we can compare configuration Cn' and $\underline{Cn_5}$ to prove that $Cn' \mathcal{R} \underline{Cn_5}$.

To begin with, the oldest activity, α, that corresponds to COG α (COG α exists by Lemma 1), contains now in its store a copy of the new object o corresponding to object i'_β in ABS. Objects o and i'_β are equivalent because first, o contains the required additional fields: cog that points the new activity β, and $myId$ with value i'; other fields are only meaningful in β. Concerning the current request, first, the local variables in MultiASP contain two more variables, but the existing ones did not change over the process. Second, the remaining statements fall in the second case of the definition of statement equivalence. Like the previous case,

[16] note that i' is necessarily a fresh identifier in β

we have:
$s_{\text{ABS}} = (x = i'\_\beta; s) \ \wedge$
$s_{\text{MultiASP}} = (x = no; [\![s]\!])$ *with* $i'\_\beta \approx^{Cn'}_{\sigma''} [\![no]\!]_{(\sigma''+\ell_4)}$.
Since $[\![no]\!]_{(\sigma''+\ell_4)} = o$ and since we showed before that o is equivalent to i'_β, we can conclude that remaining statements are equivalent.

Then, we have a new activity in MultiASP which corresponds to the fresh COG β in ABS. The content of this activity reflects the freshly created object o'. Concerning the object value, we must prove equivalence of object parameters other than *cog* and *myId*[17]. We have $\overline{v} = [\![\overline{e}]\!]^A_{(a+l)}$ and $(\overline{v_2}, \sigma_3) = \text{rename}_{\sigma'}([\![\overline{e}]\!]_{(\sigma''+\ell_4)}, \text{serialise}([\![\overline{e}]\!]_{(\sigma''+\ell_4)}, \sigma''))$. By Lemma 3 and Lemma 4 we have $\overline{v} \approx^{Cn'}_{\sigma_3} \overline{v_2}$. This ensures line 3 of definition of $\mathcal{R}$. In ABS, the new object is **`idle`** with no pending request. Line 4 and 5 of definition of equivalence $\mathcal{R}$ is verified for them because no request is currently served in ABS, and in MultiASP no *execute* request is currently served.

Finally, we have two additional futures in MultiASP concerning *freshId* and *register*, but they are not considered in the equivalence as they do not correspond to an *execute* request.

• Case of [RENDEZ-VOUS-COMM] rule
We start from the following ABS RENDEZ-VOUS-COMM reduction rule, where $Cn \xrightarrow{\text{A}} Cn'$:

$$\frac{f = \text{fresh}(\ ) \qquad i'\_\beta = [\![e]\!]^A_{(a+l)} \qquad \overline{v} = [\![\overline{e}]\!]^A_{(a+l)} \qquad p'' = \text{bind}(i'\_\beta, f, m, \overline{v}, \texttt{class}(i'\_\beta))}{\begin{array}{c} ob(i\_\alpha, a, \{l \mid x = e!\texttt{m}(\overline{e}); s\}, q)\ ob(i'\_\beta, a', p', q') \\ \xrightarrow{\text{A}} ob(i\_\alpha, a, \{l \mid x = f; s\}, q)\ ob(i'\_\beta, a', p', q' \cup p'')\ fut(f, \bot) \end{array}}$$

In the following, we only deal with the case where the communication is not performed on the caller itself, i.e. where $\alpha \neq \beta$ ($\alpha = \beta$ is similar but requires a specific proof instance).
First of all, by Lemma 1, $\exists\ \alpha\ .\ cog(\alpha, i\_\alpha) \in Cn$ and $\exists\ \beta\ .\ cog(\beta, i'\_\beta) \in Cn$.
By definition of equivalence $\mathcal{R}$ and of translational semantics, there exist $o_\alpha, \sigma_\alpha, p, Rq, \ell, \ell'', s'', o_\beta, \sigma_\beta, p', Rq$ such that the following terms belong to $\underline{\text{Cn}}$, where $Cn\ \mathcal{R}\ \underline{Cn}$:

$$\begin{array}{l} \text{ACT}(\alpha, o_\alpha, \sigma_\alpha, \{q_A \mapsto \{\ell | t = e.cog(); id = e.myId(); x = t.execute(id, \texttt{m}, \overline{e}); [\![s]\!]\} :: \\ \qquad \{\ell'' \mid s''\}\} \uplus p, Rq) \\ \text{ACT}(\beta, o_\beta, \sigma_\beta, p', Rq') \end{array}$$

with an object equivalent to i'_β in MultiASP resulting from the evaluation of e in activity α. This object has two copies: one in activity α and one in activity β, but the value of its fields in α are meaningless. Indeed, the copy of this object in α acts as a proxy to its meaningful counterpart in β. More formally, we have:

$$\begin{array}{l} o' \mapsto [cog \mapsto \beta, myId \mapsto i', [\overrightarrow{x \mapsto v'}]] \in \sigma_\alpha \quad \text{with} \quad [\![e]\!]_{\sigma+\ell} = o' \\ o'' \mapsto [cog \mapsto \beta, myId \mapsto i', [\overrightarrow{x \mapsto v''}]] \in \sigma_\beta \quad \text{with} \quad a' \approx_{\sigma_\beta} [\overrightarrow{x \mapsto v''}] \end{array}$$

[17] Recall that it is the instance of the parameters hosted in β that are meaningful.

MultiASP configuration $\underline{Cn}$ can be reduced using local rules until a configuration $\underline{Cn'}$ contains the *execute* remote method invocation as current statement. More formally, we have $\underline{Cn} \rightarrow^* \underline{Cn'}$, where the active thread of activity α in $\underline{Cn'}$ is: $\{\ell' | x = t.execute(id, \mathtt{m}, \overline{e}); [\![s]\!]\} :: \{\ell'' \mid s''\}$, with $\ell' = \ell[t \mapsto \beta, id \mapsto i']$. Then, we can reduce $\underline{Cn'}$ to a configuration $\underline{Cn''}$, $\underline{Cn'} \rightarrow \underline{Cn''}$, by the MultiASP INVK-ACTIVE rule:

$$\frac{[\![\overline{e}]\!]_{(\sigma_\alpha + \ell')} = \overline{v} \qquad f, o_f \ \text{ fresh} \qquad \begin{array}{c}[\![t]\!]_{(\sigma_\alpha+\ell')} = \beta \\ (\overline{v_r}, \sigma_r) = \text{rename}_{\sigma_\beta}(\overline{v}, \text{serialise}(\overline{v}, \sigma_\alpha))\end{array}}{\begin{array}{c}\text{ACT}(\alpha, o_\alpha, \sigma_\alpha, \{q_A \mapsto \{\ell' \mid x = t.execute(id, \mathtt{m}, \overline{e}); [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \\ \text{ACT}(\beta, o_\beta, \sigma_\beta, p', Rq') \\ \rightarrow \text{ACT}(\alpha, o_\alpha, \sigma_\alpha[o_f \mapsto f], \{q_A \mapsto \{\ell' \mid x = o_f; [\![s]\!]\} :: \{\ell'' \mid s''\}\} \uplus p, Rq) \\ \text{ACT}(\beta, o_\beta, \sigma_\beta \cup \sigma_r, p', Rq' :: (f, execute, i', \mathtt{m}, \overline{v_r}))\ \text{FUT}(f, \bot)\end{array}}$$

We finally have to prove that $Cn' \ \mathcal{R} \ \underline{Cn''}$. First, we have as many terms that changed in Cn' as in $\underline{Cn''}$: there are two *ob* and ACT terms and one added future in configurations of both side.

Concerning the fresh future, it is the same in the two configurations: we conclude by line 6 of the definition of equivalence $\mathcal{R}$.

Concerning activity α; the local variables remain unchanged except temporary variables. The element that actually changed is the current statement. We have $(x = f; s) \approx^{Cn'}_{\sigma_\alpha} (x = o_f; [\![s]\!])$ by the definition of the equivalence between statement (second case); we need $f \approx^{Cn'}_{\sigma_\alpha} [\![o_f]\!]_{\sigma_\alpha + \ell'}$ which is true because $[\![o_f]\!]_{\sigma_\alpha+\ell'} = o_f$ and $\sigma_\alpha(o_f) = f$; thus $f \approx^{Cn'}_{\sigma_\alpha} \sigma(o_f)$ by definition of equivalence. We conclude the equivalence between activities α using line 4 of the definition of equivalence $\mathcal{R}$.

Concerning activity β, a new pending request is created in MultiASP configuration $\underline{Cn''}$, corresponding to the new inactive thread in the ABS configuration Cn'. According to line 5 of the definition of equivalence $\mathcal{R}$, we have $\{l|s\} \in q' \wedge l(destiny) = f$ with $\{l|s\} = \text{bind}(i'\_\beta, f, m, \overline{v}, \mathtt{class}(i'\_\beta))$; there is only one case to consider because we know that the request is not served yet. Thus, we have to prove:

$$\begin{array}{l}\exists i', \mathtt{m}, \overline{v_r}, \ell_\beta, s'.(f, execute, i', \mathtt{m}, \overline{v_r}) \in Rq \\ \wedge o_\beta.retrieve(i') = o \wedge \text{bind}(o, \mathtt{m}, \overline{v_r}) = \{\ell_\beta | s'\} \\ \underline{\text{with}} \quad \forall x \in dom(l) \backslash destiny.\, l(x) \approx^{Cn'}_{\sigma_\beta} \ell_\beta(x) \wedge s \approx^{Cn'}_{\sigma_\beta + \ell_\beta} s'\end{array}$$

Firstly, we have indeed the new request in queue in MultiASP configuration Cn''. We now need to ensure that $o_\beta.retrieve(i') = o$ which is guaranteed by the **Invariant Reg**. On the other hand, the result of bind in MultiASP is similar to the one in ABS. The local variables on ABS side also appear on MultiASP side equivalently, except the two following points:

- There is no *destiny* variable in MultiASP. However, this is taken into account by the definition of equivalence $\mathcal{R}$, which compares the *destiny* variable with the future of the corresponding MultiASP request.

– The transmitted parameters $\overline{v_r}$ are the copies of the method parameters in ABS. Ensuring equivalence between request parameters in this case is handled by Lemma 3 for obtaining the equivalence of the emitted values, and by Lemma 4 to ensure that, once values have been copied to β and renamed, equivalence is still ensured.

• Case of [RETURN] rule

We start from the following RETURN ABS reduction rule, such that $Cn \xrightarrow{\text{A}} Cn'$:

$$\frac{v = [\![e]\!]^{A}_{(a+l)} \quad f = l(\text{destiny})}{ob(i\_\alpha, a, \{l \mid \texttt{return}\ e; s\}, q)\ fut(f, \bot) \xrightarrow{\text{A}} ob(i\_\alpha, a, \texttt{idle}, q)\ fut(f, v)}$$

To begin with, we have:

– By Lemma 1, $\exists cog(\alpha, i\_\alpha) \in Cn$.

– By definition of equivalence $\mathcal{R}$:

- $\exists o_\alpha, \sigma, p, Rq$ such that $\text{ACT}(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$.
- $\exists f, i, m, \overline{v''}, \ell', s', \ell'', s''$ such that $(f, execute, i, m, \overline{v''})_A \mapsto \{\ell' \mid s'\} :: \{l'' \mid s''\} \in p$, and $\texttt{return}\ e; s \approx^{Cn}_{\sigma+\ell'} s'$. By definition of equivalence between statements, we have $s' = (\texttt{return}\ e; [\![s]\!])$ since the first statement is not in the form of $x = e; s$.
- by line 7, we also have $\text{FUT}(f, \bot) \in \underline{Cn}$.

Furthermore, we know that in our translation, a method call on an object is always wrapped in an *execute* method call on an active object.

Therefore, we have $s'' = (x = \bullet; \texttt{return}\ x)$.

Thus, we find the following MULTIASP configuration $\underline{Cn}$ such that $Cn\mathcal{R}\underline{Cn}$:

$$\begin{array}{l}\text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell' \mid \texttt{return}\ e; [\![s]\!]\} :: \{\ell'' \mid x = \bullet; \texttt{return}\ x\}\} \uplus\ p', Rq)\\ \text{FUT}(f, \bot)\end{array}$$

From $\underline{Cn}$, we can apply the RETURN-LOCAL MULTIASP reduction rule to have configuration $\underline{Cn'}$ such that $\underline{Cn} \to \underline{Cn'}$:

$$\frac{v' = [\![e]\!]_{\sigma+\ell'}}{\underline{Cn} \to \text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell'' \mid x = v'; \texttt{return}\ x\}\} \uplus p', Rq) \quad \text{FUT}(f, \bot)}$$

Then, after applying the ASSIGN-LOCAL MULTIASP reduction rule on $\underline{Cn'}$, we fall in configuration $\underline{Cn''}$ such that $\underline{Cn'} \to \underline{Cn''}$ as follows:

$$\begin{array}{l}\text{ACT}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell''[x \mapsto v'] \mid \texttt{return}\ x\}\} \uplus\ p', Rq)\\ \text{FUT}(f, \bot)\end{array}$$

We now denote $\ell_3 = \ell''[x \mapsto v']$. From $\underline{Cn''}$, we can apply the RETURN MULTIASP reduction rule to end up in configuration $\underline{Cn'''}$ such that $\underline{Cn''} \to \underline{Cn'''}$:

$$\frac{[\![x]\!]_{\sigma+\ell_3} = v'}{\underline{Cn''} \to \text{ACT}(\alpha, o_\alpha, \sigma, p', Rq) \quad \text{FUT}(f, v', serialise(v', \sigma))}$$

We further denote $\sigma_s = serialise(v', \sigma)$. Recall $[\![e]\!]_{\sigma+\ell'} = v'$. Now, we have to check $Cn'\ \mathcal{R}\ \underline{Cn'''}$. Concerning the resolved future, we have the future term f in

both ABS and MultiASP configurations. We have to ensure that $v \approx_{\sigma_s}^{Cn'} v'$, and for that we have to ensure that $v \approx_{\sigma}^{Cn'} [\![e]\!]_{\sigma+\ell'}$. Indeed, by Lemma 3, we have $[\![e]\!]_{a+l}^{A} \approx_{\sigma}^{Cn'} [\![e]\!]_{\sigma+\ell'}$ and thus $v \approx_{\sigma}^{Cn'} v'$ and thus by Lemma 4 $v \approx_{\sigma_s}^{Cn'} v'$.

Concerning activity α, the request corresponding to future f is no more existing, which matches with the fact that the ABS current task becomes **`idle`**: in the definition of equivalence, there is one less request that is to be compared on line 4 of Figure 12.

The other elements of the activity did not change, thus preserving their equivalence.

• Case of [Skip], [Cond-True], [Cond-False], [Context], [Cog-Sync-Call], [Self-Sync-Call], [Cog-Sync-Return-Sched], [Self-Sync-Return-Sched] and [Rem-Sync-Call] rules

To finish, none of those rules above are considered in the proof of Theorem 3, for different reasons. We review the reasons for not treating each of them below.

Firstly, Skip, Cond-True, Cond-False and Context rules are kept unchanged from the translation provided by the original Java backend for ABS. Consequently, they are already supposed to be sound and correct, and their proof is anyway trivial. Furthermore, the **`await`** statement on conjunctive guards and boolean expressions are not handled here. In fact, they do not raise a particular difficulty because they only involve local computation.

Secondly, all the rules dealing with local synchronous method calls, namely Cog-Sync-Call, Self-Sync-Call, Cog-Sync-Return-Sched and Self-Sync-Return-Sched are not considered in the proof. Indeed, those rules send back the current task in queue and schedule it right away with the continuation statement **`cont`** to fake a synchronous treatment. Instead, we rely on a traditional stack of method calls and avoid considering the **`cont`** keyword. Besides, these rules make use of statements that are treated in other cases, like **`await`** and **`get`**. Consequently, considering these rules would be redundant and would not bring any particular insight on the correctness of the translation. Additionally, the practical translation of those aspects is again unchanged from the original Java backend for ABS.

Thirdly, in the case of the rule for remote synchronous method calls, Rem-Sync-Call, the ABS rule is a composition of a remote asynchronous method call and a future read; in the translation, we directly inline this composition. Thus, the proof for this rule simply inlines the two proofs for Rendez-vous-Comm and Read-Fut rules.

In conclusion, all reduction rules of ABS ensure that an equivalent final configuration is reachable in MultiASP, which completes the proof of Theorem 3. □

Overall, although equivalence of ABS and MultiASP configurations is achieved for all ABS reduction rules, it is impossible to have strong simulation here, because of the intrinsic differences between the two languages. We must consider

ABS rule	MultiASP **rule**	**Additional** MultiASP **rules**
Assign-Local	Assign-Local	–
Assign-Field	Assign-Field	–
Await-True	–	Update / – , Invk-Passive, Return-Local Assign-Local-Tmp
Await-False	Invk-Future	–
Release-Cog	–	–
Activate	–	Activate-Thread / (Serve, Invk-Passive,Return-Local, Assign-Local-Tmp)
Read-Fut	–	Set-Hard-Limit, Update / – , Invk-Passive, Return-Local, Set-Soft-Limit, Assign-Local-Tmp
New-Object	New-Object	Invk-Passive, Assign-Local-Tmp, Return-Local
New-Cog-Object	New-Object	New-Active, Assign-Local-Tmp, Invk-Active-Meta, Return
Rendez-vous-Comm	Invk-Active	Invk-Passive, Return-Passive, Assign-Local-Tmp
Return	Return	Return-Local, Assign-Local-Tmp

Table 1: Summary table of ABS to MultiASP simulation. Assign-Local-Tmp means Assign-Local on a variable introduced by the translation whether other Assign-Local labels means assignmetns that target ABS local variables. Invk-Active-NE means Invk-Active on a method that is not execute whether other Invk-Active labels means invocation of an execute request.

some of the ABS and some of the MultiASP rules as *silent actions*. In particular, we have indeed some ABS reduction rules that are strictly associated to one MultiASP reduction rule, like for strong simulation. For some other ABS reduction rules, there is an associated MultiASP one, but additional silent rules are also needed to reach the correct simulation. On the other hand, some ABS reduction rules cannot be simulated by MultiASP ones, or more precisely, they strictly correspond to no MultiASP rule and either equivalence is just maintained, or some non-observable transitions must be achieved. We denote those cases as – in Table 1 that summarises our results on simulation. The / character indicates an option depending on a criteria detailed in the proof but not in the table.

Further comments can be made about this table. First, we can note that additional Invk-Passive and Assign-Local-Tmp MultiASP rules are quite omnipresent. Indeed, we exhibited before that our translation introduces additional communications and temporary variables. We can notice that, if we ignore the introduction of those harmless rules, most of the important ABS rules can be simulated by a single MultiASP one. Second, we can simulate Await-False

ABS rule with Invk-Future MultiAsp rule, but the Await-True rule has no MultiAsp equivalent, then it might have been more coherent to consider Await-False as silent too. Third, New-Object and New-Cog-Object are simulated by New-Object MultiAsp rule but we can distinguish the two cases. Indeed, the New-Object MultiAsp rule should distinguish whether the cog is empty or not. More precisely, New-Cog-Object ABS rule is simulated by a New-Object MultiAsp on an empty cog, while New-Object ABS rule is simulated by a New-Object MultiAsp rule on a non-empty cog. There is thus no ambiguity in the simulation. From another point of view, New-Active MultiAsp rule could be the visible rule to simulate New-Cog-Object ABS rule instead of New-Object MultiAsp rule, but doing so would mean that we create an (active) object in MultiAsp that has no equivalent object in ABS; thus we prefer tracing New-Object rules on MultiAsp side. Also for this simulation, note that the additional Invk-Active-Meta on MultiAsp side is invisible because it only applies to meta-requests that do not exist in ABS, such as *register* and *freshId*. Symmetrically, the visible Invk-Active MultiAsp rule for simulating Rendez-vous-Comm ABS rule only corresponds to *execute* requests which are the ones that are visible in ABS. Again, we can easily distinguish request of Invk-Active rules, so there is no ambiguity in the simulation.

Silent MultiAsp *actions.* Finally, we can now easily list the MultiAsp reduction rules that are not visible in the simulation: New-Active, Serve, Invk-Passive, Return-Local, Update, Activate-Thread, Set-Hard-Limit, Set-Soft-Limit Invk-Active on a method that is not execute (Invk-Active-Meta), Assign-Local on an intermediate variable introduced by the translation (Assign-Local-Tmp). The other MultiAsp reduction rules are the ones that are observable in the simulation. This gives a particular insight on the construction of active object-based languages and highlight possible shortcomings in language designs.

Silent ABS actions. As shown by the table, the silent ABS rules are: Await-True, Release-Cog, Activate, Read-Fut. It is interesting to note that those rules concern part of the future manipulation, as highlighted above, and some part of the thread manipulation that is handled in a different way in the ABS semantics. For example, a single MultiAsp rule is necessary for an object to release a thread whether two rules are necessary in ABS.

C.5 Core Proof of Theorem 4 - From MultiAsp to ABS

In this section, we prove Theorem 4, namely that MultiAsp translation corresponds to a valid ABS execution.

Theorem 4. *Any reduction of the* MultiAsp *translation corresponds to a valid ABS execution:* $\underline{Cn_0} \rightarrow^* \underline{Cn} \Rightarrow \exists Cn.\, Cn_0 \overset{\mathrm{A}}{\rightarrow}^* Cn \wedge Cn\, \mathcal{R}\, \underline{Cn}$

In this direction, the main difficulty is that we introduced additional steps in the reduction. On the other hand, the languages have only few concurrent

rules (method invocation and future awaiting) and, as there can be a single active thread at a time, it prevents any local concurrency. Thus, the sequence of additional actions never performs a wait-by-necessity, it runs until the end without any observable interruption. Only thread activation, thread passivation, method invocation, and future update can create interleavings.

Proving this direction is a little trickier as the translated code is more operational; we can be in intermediate states in MultiASP that must be attached to the right state in ABS. Here, one should first observe that the translation of an ABS primitive mostly involves a single action that impacts the state of the equivalent ABS program. For example, for remote invocations, solely the call to *execute* impacts the request queue of the destination and has an effect on the equivalence relation. The global idea is to see only those actions and to enable statement translation to do the same. In the case of *execute*, assignments to intermediate variables are ignored in the equivalence, and all the states that precede the execution of the remote method invocation are considered equivalent to the same ABS state. This is mostly handled by ignoring adequately assignments to variables that are not part of the ABS code. Also one should notice that there is an intermediate state when the request is served but the associated ABS method is not started yet, one should also consider this case in the notion of equivalence: such a just started request is similar to the case when the request still is in the queue. It is also important to notice that the theorem needs no restriction on futures which value is a future because when a future access is possible in MultiASP it can be faithfully simulated in ABS: ABS allows more control on the status of futures.

The goal here is not to do the exhaustive simulation proof mostly because the technical details will be massively redundant with the proof of Theorem 3. However, what is important to show here is that the translation ensures that no additional behaviour can be introduced by the MultiASP semantics. We provide a summary of the most important points of the proofs, i.e. the refinement of the equivalence relation presented in the paper, and a summary of the equivalence between reductions.

Adapting the equivalence relation To prove that all the MultiASP executions faithfully respect the ABS semantics, the equivalence relation needs to be refined, mostly to account for the fact that several statements are used in MultiASP to simulate a single ABS statement. In practice, it is easy to identify one statement that will trigger the same rule on both sides whilie the others are sintermediate steps generally preparing the main statement (e.g. fetching the identifier and cog of the target object of an invocation). We thus define a new equivalence relation on statements in Figure 13.

The first and the last rule are unchanged, but additional rules allow to discard **setLimitHard** instructions (the thread having just performed a **setLimitHard** is considered as still in the same state even if one statement is missing). Similarly, assignment to temporary variables can be added or removed, but when removing an assignment to a temporary variable one can use the value assigned if it is a

$$s \approx^{Cn}_{(\sigma+\ell)} [\![s]\!] \qquad \frac{s \approx^{Cn}_{(\sigma+\ell)} (\mathbf{setLimitHard}; s')}{s \approx^{Cn}_{(\sigma+\ell)} s'}$$

$$\frac{s \approx^{Cn}_{(\sigma+\ell)} (tmp = z; s') \quad tmp \text{ is a local variable introduced by the translation}}{s \approx^{Cn}_{(\sigma+\ell)} s'}$$

$$\frac{s \approx^{Cn}_{(\sigma+\ell)} s' \quad tmp \text{ is a local variable introduced by the translation}}{s \approx^{Cn}_{(\sigma+\ell)} (tmp = z; s')}$$

$$\frac{s \approx^{Cn}_{(\sigma+\ell+(tmp \mapsto [\![e]\!]_{(\sigma+\ell)}))} s' \quad tmp \text{ is a local variable introduced by the translation}}{s \approx^{Cn}_{(\sigma+\ell)} (tmp = e; s')}$$

$$\frac{s = (x = v; s_1) \qquad s' = (x = e; [\![s_1]\!]) \qquad v \approx^{Cn}_{\sigma} [\![e]\!]_{(\sigma+\ell)}}{s \approx^{Cn}_{(\sigma+\ell)} s'}$$

Fig. 13: New equivalence on statements for MultiASP to ABS proof

simple expression. This will be useful to relate $no = o; s; x = no$ in MultiASP with $x = o$ in ABS.

Two other modifications are to be made concerning the equivalence on configurations (Figure 12 of the paper):

- When considering the case from ABS to MultiASP, we had as many COG as active objects. However, in ABS, when a COG is created, it is populated with at least one object, whereas in the translation of `new`, the instruction creating an active object (`newActive`) is separated from the creation of the new object populating the COG in ABS. Even if, in any case, the thread cannot be interrupted when doing those steps, this creates a point where the equivalence relation is not verified anymore. Such a situation is ruled out by considering, in line 1 of Figure 12, only activities that have at least one object inside. In other words, activities only populated with a COG active object are considered not to be part of the configuration yet. This can easily be formulated by looking at the content of the local store of the activity: if σ only contain the COG object and no object corresponding to an ABS object, then the activity should not be considered in the equivalence. More formally: *Line 1 of Figure 12 is modified as follows:* $act(\alpha, o_\alpha, \sigma, p, Rq) \in \underline{Cn}$ *should be added the side condition:* σ *has more than one entry.*
- There is an intermediate state where the request has been served in MultiASP but the ABS method has not yet started to be executed (i.e. when the stack corresponding to this request has a single entry). In this case the corresponding thread is necessarily the active one. This should be considered

similarly to requests still in queue: *In Line 4 of of Figure 12, one more case is necessary:*

$$\big((f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge\ \ell'(\texttt{this}) = o\big)$$

should be replaced by

$$\big((f, execute, i, m, \overline{v''})_A \mapsto \{\ell'|s'\} :: \{\ell''|s''\} \in p \wedge\ \ell'(\texttt{this}) = o \vee$$
$$(f, execute, i, m, \overline{v''})_A \mapsto \{\ell''|s''\} \in p \wedge o_\alpha.retrieve(i) = o \wedge bind(o, m, \overline{v''}) = \{\ell'|s'\}\big)$$

Modulo those minor changes, the proof is similar to the case before, though a little more tedious. We only provide the summarising table of the reductions with comments.

Proof principles Like in the situation where we go from ABS to MultiASP, a notion of observability is to be defined, to focus on what is meaningful to consider for equivalence. In our sense, the most important step to be observable is request emission. Indeed, it characterises by its own the available objects and the available COG, the communication and the variables and statements that allowed to perform it.

Let us first recall the theorem:

Theorem 4 Any reduction of the MultiASP translation corresponds to a valid ABS execution: $\underline{Cn_0} \rightarrow^* \underline{Cn} \Rightarrow \exists Cn. Cn_0 \overset{\text{A}}{\rightarrow}^* Cn \wedge Cn\,\mathcal{R}\,\underline{Cn}$

Note that we do not need a restriction like $\nexists f, f'. fut(f, f') \in Cn'$ in this direction because fetching the value of a future which is itself a future is possible in ABS but would block in ASP. Some ABS reductions could not be matched in ASP and potentially lead to a blocked state, but when the reduction is possible in MultiASP it can be faithfully simulated in ABS, even in this case.

Table 2 summarises our results on how the MultiASP translated program can be simulated by a valid ABS execution. Further comments can be made on the rules. First ASSIGN-LOCAL and ASSIGN-FIELD raise no particular comment, except for the assignment of intermediate variables introduced by the translation that can be safely ignored.

INVK-FUTURE only activates if the current limit is soft, which only happens in the translation of the *await* statement and if the future is not yet resolved. Indeed, no additional future can be awaited in the code corresponding to the translation. The COG is additionally released.

In the case of INVK-PASSIVE, two cases are possible. The corresponding ABS rule can be READ-FUT if the invoked method is *get* and the current limit is a hard limit; or, it is AWAIT-TRUE if the invoked method is *get* and the current limit is a soft limit. It is silent for all invocations of methods that are internal to the translation (remember that the passive method invocation in ABS is not considered in the proof).

Distinction between ACTIVATE-THREAD and SERVE is made depending on the status of the request (whether it has started to be served or not) in MultiASP. Such a distinction does not exist in ABS.

MultiASP **rule**	**ABS rule**	**Additional ABS rules**
ASSIGN-LOCAL	ASSIGN-LOCAL	–
ASSIGN-LOCAL-TMP	–	–
ASSIGN-FIELD	ASSIGN-FIELD	–
INVK-FUTURE	AWAIT-FALSE	RELEASE-COG
INVK-PASSIVE	–	–/READ-FUT/AWAIT-TRUE
ACTIVATE-THREAD	ACTIVATE	–
SERVE	ACTIVATE	–
INVK-ACTIVE	RENDEZ-VOUS-COMM	–
INVK-ACTIVE-META	–	–
NEW-OBJECT in an activity with no ABS object	NEW-COG-OBJECT	–
NEW-OBJECT in a non-empty activity	NEW-OBJECT	–
NEW-ACTIVE	–	–
RETURN-LOCAL	–	–
RETURN	RETURN	–
UPDATE	–	–
SET-SOFT-LIMIT	–	–
SET-HARD-LIMIT	–	–

Table 2: Summary table of MultiASP to ABS simulation. Like previously, ASSIGN-LOCAL-TMP means ASSIGN-LOCAL on a variable introduced by the translation whereas other ASSIGN-LOCAL labels means assignments that target ABS local variables. INVK-ACTIVE-META means INVK-ACTIVE on a method that is not *execute* whereas other INVK-ACTIVE labels means invocation of an *execute* request.

INVK-ACTIVE reductions that send an execute request exactly match the RENDEZ-VOUS-COMM rule. Note that here, the preliminary steps and additional steps performed in the translation are handled by the equivalence relation, those steps also ensure that the object exist in ABS and can be accessed through its COG. Of course, INVK-ACTIVE reduction for other methods (labeled INVK-ACTIVE-META) have no corresponding reduction in ABS.

Object creation NEW-OBJECT can either correspond to the creation of an object in the same COG (NEW-OBJECT) or in a new COG if the store of the activity where the object is created is empty (only contains the COG object).

Assignment of a result to a future (RETURN rule) is matched exactly except that the request in MultiASP is an "execute" request.

Silent ABS actions. ABS reduction rules AWAIT-TRUE, READ-FUT, RELEASE-COG are not visible in MultiASP.

Silent MultiASP *actions.* Finally, the MultiASP reduction rules that are not visible in the simulation are the following: NEW-ACTIVE, INVK-PASSIVE, RETURN-LOCAL, UPDATE, SET-HARD-LIMIT, SET-SOFT-LIMIT INVK-ACTIVE on a method

that is not execute (Invk-Active-Meta), Assign-Local on an intermediate variable introduced by the translation (Assign-Local-Tmp). Except Invk-Passive none of them needs any reduction on the ABS side: the reduced configuration is still equivalent to the same original ABS configuration.

D Experimental Evaluation

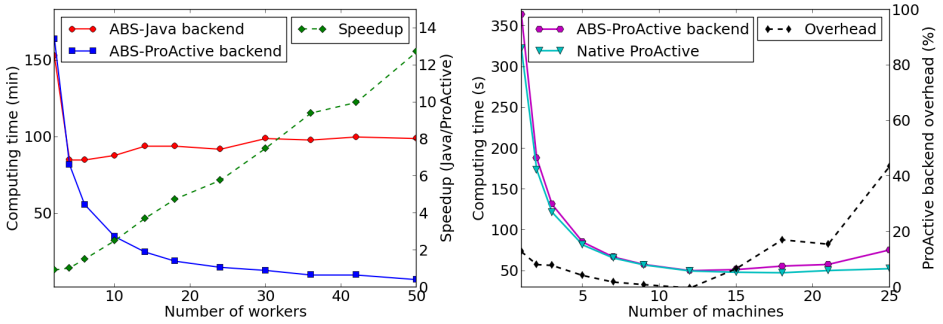
In order to test the ProActive backend, we first use four example programs given in the ABS tool suite. These examples are: a *Bank account* program that consists of 167 lines of ABS and that creates 3 cog; a *Leader election* algorithm over a ring (62 lines, 4 cog); a *Chat* application (324 lines, 5 cog), and a *Deadlock* example that hangs by circular dependencies between activities (69 lines, 2 cog). For all examples, we observe that the behavior of the program translated with the ProActive backend is the same as the one translated with the Java backend. Thus, the scheduling policy enforced in ProActive faithfully respects the one of the reference implementation. These examples run in a few milliseconds; they are inadequate for performance analysis but they allow us to check that the ProActive backend behaves correctly.

We have also conducted an experimental performance evaluation of an ABS distributed application translated with the ProActive backend and distributed on a cluster. We compare it to the performance of the same ABS application translated with the Java backend, run on a single machine. The considered use case is the pattern matching of a DNA sequence using the MapReduce programming model Map instances (workers) are created in their own cog to make them work in parallel. We consider a searched pattern of 250 bytes, and a database of 5 MB of DNA sequences. Each map searches for the maximum matching sequence of a chunk. Then, a reducer outputs the global maximum matching sequence. We compute the execution time of the whole use case when varying the number of workers for both generated applications. In the case of ProActive translation, the number of workers is twice the number of physical machines used (two workers run on each machine). In the case of the Java translation, all workers run in parallel on a single machine (no support for distribution). All used machines have 2 dual core CPUs at 2.6GHz, and 8 GB of RAM[18].

Figure 14a shows execution time of both ProActive and Java translations of the ABS application from 2 to 50 workers, therefore using from 1 to 25 physical machines with ProActive. The execution time of the application stemming from the ProActive backend is sharply decreasing for the first added machines and then decreases at a slower rate. On the other hand, the application stemming from the Java backend has an optimal degree of parallelism of 4 workers (which actually is the number of cores of the machine) and then cannot benefit from higher parallelism; execution time even increases afterwards due to context switching overhead. On the opposite, increasing the degree of parallelism for

[18] We use a cluster of Grid5000 platform: `http://grid5000.fr`

[19] Each measurement is an average of five executions.



(a) Against existing solution (b) Against native code

Fig. 14: Execution time of DNA-matching ABS application[19]

the application stemming from the ProActive backend gives a linear speedup, because it balances the load between machines.

Figure 14b compares now the performance of the generated ProActive code to a hand-written version. In the generated program, we have manually replaced the translation of functional ABS types (integers, booleans, lists, maps) with corresponding standard Java types. Indeed, our point is to evaluate the additional communication cost of our translation, not the performance of ABS types compared to standard Java types In this context, we can see that the overhead introduced by the ProActive backend is rather low since it is generally kept under 10%. At the biggest stage, the application translated with the ProActive backend starts to have a higher overhead because it involves too much communication. In conclusion, thanks to the ProActive backend, we were able to turn a local ABS application into a high-performance distributed application. In addition, the code generated by the ProActive backend maintains a low overhead compared to a native solution.

Acknowledgements. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).